A PROCESS ELABORATION FORMALISM FOR WRITING AND ANALYZING PROGRAMS

University of Southern California

PREPARED FOR
Defense Advanced Research Projects Agency

October 1975

KEEP UP TO DATE

Between the time you ordered this report—which is only one of the hundreds of thousands in the NTIS information collection available to you—and the time you are reading this message, several *new* reports relevant to your interests probably have entered the collection.

Subscribe to the **Weekly Government Abstracts** series that will bring you summaries of new reports *as soon as they are received by NTIS* from the originators of the research. The WGA's are an NTIS weekly newsletter service covering the most recent research findings in 25 areas of industrial, technological, and sociological interest—invaluable information for executives and professionals who must keep up to date.

The executive and professional information service provided by NTIS in the **Weekly Government Abstracts** newsletters will give you thorough and comprehensive coverage of government-conducted or sponsored research activities. And you'll get this important information within two weeks of the time it's released by originating agencies.

WGA newsletters are computer produced and electronically photocomposed to slash the time gap between the release of a report and its availability. You can learn about technical innovations immediately—and use them in the most meaningful and productive ways possible for your organization. Please request NTIS-PR-205/PCW for more information.

The weekly newsletter series will keep you current. But *learn what you have missed in the past* by ordering a computer **NTISearch** of all the research reports in your area of interest, dating as far back as 1964, if you wish. Please request NTIS-PR-186/PCN for more information.

WRITE: Managing Editor
5285 Port Royal Road
Springfield, VA 22161

Keep Up To Date With SRIM

SRIM (Selected Research in Microfiche) provides you with regular, automatic distribution of the complete texts of NTIS research reports *only* in the subject areas you select. SRIM covers almost all Government research reports by subject area and/or the originating Federal or local government agency. You may subscribe by any category or subcategory of our WGA **(Weekly Government Abstracts)** or **Government Reports Announcements and Index** categories, or to the reports issued by a particular agency such as the Department of Defense, Federal Energy Administration, or Environmental Protection Agency. Other options that will give you greater selectivity are available on request.

The cost of SRIM service is only 45¢ domestic (60¢ foreign) for each complete microfiched report. Your SRIM service begins as soon as your order is received and processed and you will receive biweekly shipments thereafter. If you wish, your service will be backdated to furnish you microfiche of reports issued earlier.

Because of contractual arrangements with several Special Technology Groups, not all NTIS reports are distributed in the SRIM program. You will receive a notice in your microfiche shipments identifying the exceptionally priced reports not available through SRIM.

A deposit account with NTIS is required before this service can be initiated. If you have specific questions concerning this service, please call (703) 451-1558, or write NTIS, attention SRIM Product Manager.

This information product distributed by

NTIS U.S. DEPARTMENT OF COMMERCE
National Technical Information Service
5285 Port Royal Road
Springfield, Virginia 22161

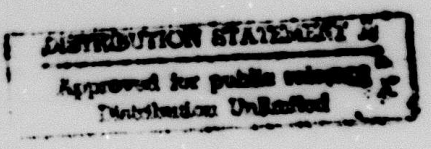
ARPA ORDER NO. 2223

ISI RR-75-35

October 1975

David Wilczynski

A Process Elaboration Formalism for Writing and Analyzing Programs



INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/Marina del Rey/California 90291

(213) 822-1511

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ISI/RR-75-35	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* A Process Elaboration Formalism for Writing and Analyzing Programs		5. TYPE OF REPORT & PERIOD COVERED Research 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) David Wilczynski		8. CONTRACT OR GRANT NUMBER(s) DAHC 15 72 C 0308
9. PERFORMING ORGANIZATION NAME AND ADDRESS USC/Information Sciences Institute 4676 Admiralty Way Marina del Rey, CA 90291		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order #2223 Program Code 3D30 & 3P10
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE October 1975 13. NUMBER OF PAGES<br~~116~~ 117
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* ------		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT *(of this Report)* This document approved for public release and sale; distribution unlimited.		
17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)* -----		
18. SUPPLEMENTARY NOTES PRICES SUBJECT TO CHANGE		
19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)* automatic debugging, automatic programming, execution graphs, production systems, program intentions		
20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)* (OVER)		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

20. ABSTRACT

This research effort presents a formalism for writing programs which explicitly addresses and highlights some program construction issues. The formalism, a kind of production system, generates a graph that defines the process under inspection, making explicit both when and where variable bindings take place. From the standpoint of proper data structuring these extra dimensions are useful for analyzing a program, particularly with respect to ease of data access, access ambiguity, proper sequence of bindings, and other related issues. Because the formalism is a natural one for parsing a protocol of an instance of the process described by the productions, the system will be able to run in two modes: generation (to produce a behavior instance) or parse (determining whether a particular behavior instance could have been generated from a given program). Both these capabilities are important in debugging programs, especially those written in an Automatic Programming environment in which the system may be communicating with a nonprogrammer.

ARPA ORDER NO. 2223

ISI/RR-75-35
October 1975

David Wilczynski

A Process Elaboration Formalism for Writing and Analyzing Programs

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/Marina del Rey/California 90291
(213) 822-1511

THIS RESEARCH IS SUPPORTED BY THE ADVANCED RESEARCH PROJECTS AGENCY UNDER CONTRACT NO. DAHC15 72 C 0308, ARPA ORDER NO. 2223/1, PROGRAM CODE NO. 3D30 AND 3P10

CONTENTS

ABSTRACT

The primary goal of an Automatic Programming system is to generate programs from some high-level description of a user's problem. This task may involve a diversity of efforts, ranging from modelling the user to optimizing the final program product. In particular, the choice of a suitable internal program model will influence the direction and capabilities of an Automatic Programming system; the form of the language will have an impact not only on the ease of the translation task, but on the scope of the program analysis for determining the accuracy of the generated program as well.

This research effort presents a system called XREP, which includes a formalism for writing programs that explicitly addresses and highlights some program construction and verification issues. The formalism, a production system, includes facilities for generating an object and referencing it by specifying its class type and identifying the desired instance by providing some limiting predicate, the predominant method used in human communication for referencing objects. XREP's language interpreter generates a graph that defines the process under inspection, making explicit both when and where variable bindings for generated objects take place. From the standpoint of proper data structuring these extra dimensions found in the execution graph are useful for analyzing a program, particularly with respect to ease of data access, detecting access ambiguity, proper sequence of bindings, and other related issues.

Another facet of program writing includes the ability to test the final product in order to verify that the program's behavior matches the user's expectation. XREP accepts an intention string of observable events, externally supplied by the user, for this purpose. Because the production language formalism is natural for a parsing task, the intention string, as a protocol of an instance of the process described by the productions, can be parsed for acceptability. The system will thus be able to run in two modes: generation (to produce a behavior instance) or parse (determining whether a particular behavior instance could have been generated from a given program).

In order to show the adequacy of the various representations, particularly the production language, the execution graph, the form of the data variables and objects, and the intention string mechanism, specific automatic debugging techniques were developed that apply to problems normally found in human communications, such as improperly stated loop control, ambiguous references, and data structuring faults. The nature of this debugging effort emphasizes some of the problems which an Automatic Programming translator will face in trying to convert human inputs into a computer program.

Though this research investigates only one analytical phase of Automatic Programming, the form of the representations chosen for it has an impact upon the entire effort; the capabilities displayed in this report are meant as a showcase for those formalisms. Thus, XREP's variables, as a counterpart to natural language objects, are shown to have an integrated place within the

production language, while their placement within the execution graph promotes powerful and intuitive accessing mechanisms. The nature of the production language not only makes the execution graph simple to generate, but also associates them visually, making it easy to relate analyses in the graph to the language. The intention string provides a reasonable, if not formal, way to specify program expectations, with the production language a perfect vehicle for carrying out the associated parsing. And, finally, high-level debugging techniques are shown to be possible in a suitably rich environment.

This is part of a series of reports describing ISI research directed toward reducing significantly the cost of military software while improving its application and upgrading the general quality of software. This report covers a significant portion of the author's USC doctoral dissertation, completed at ISI.

1. PROBLEM STATEMENT

1.1 INTRODUCTION

The concept of a programming environment has added new dimensions to software research. With the advent of interactive use of computers a programmer can participate actively in software design and development. It is no longer realistic to view programming as a process of discrete steps starting at composition, then alternating between submittals and debugging the results. Instead it becomes a dynamic process with unclear demarcations. Recent programming systems specifically designed to operate interactively, the best example of which is INTERLISP [TEITELMAN 74], exemplify this concept by also taking an active role in the programming process. INTERLISP not only provides tools to the programmer, but it also "watches" over the process, giving aid when it can by detecting local errors and providing numerous "smart" commands to hide unnecessary programming details. Only a limited attempt is made, however, to "understand" the program, a task which falls into a different area of research called Automatic Programming.

The final goal of Automatic Programming is to be able to generate computer programs from natural descriptions of the tasks to be performed. By attempting to take over the entire generation process, Automatic Programming represents the ultimate extension of the capabilities of the programming environment. Because of economics and the state of our knowledge, any Automatic Programming system will fall short of that ideal in the foreseeable future. But progress in producing more capable and active Automatic Programming systems depends entirely on the ability of the researchers to understand and model the programming process. A useful programming model must cover a variety of tasks: the host of ways to specify programs, program construction issues, verification and debugging, and so forth(i). Yet this understanding is possible because the domain is limited to one of processes, programs, and algorithms. The diversity of this knowledge allows different aspects of the total problem to be researched separately. The investigation of these independent areas contributes to the long-range project, while in the short term techniques are discovered for extending existing software systems. Thus we can envision interactive Automatic Programming systems which work together with a novice to produce a program from some process description. The efforts of this dissertation are directed toward this kind of framework, i.e., a user using natural language to interact with an Automatic Programming system.

A system of this kind must have some basic knowledge, including natural language understanding, awareness of programming concepts like variables, loops, scope, structure, and debugging capabilities, all of which must be relatable to the human. The need for a progressive

---

(i) An overview of Automatic Programming can be found in [BALZER 72].

dialogue suggests that the form of internal representations should be close to the user's original in order to promote a natural basis for communication. Though the present report does not directly deal with this concept, it is the basis for many implementation decisions. The hypothetical nature of an Automatic Programming system forces any claims and assumptions to rely on intuition rather than strict results. Still, as an experimental study in representation, the results are independent of the Automatic Programming framework.

The system to be presented, called XREP, consists of a language, an interpreter, a monitor, and a debugger. The interpreter and monitor execute the program while building a representative graph which is used by the interpreter to carry out evaluations and by the debugger as a history of the process. The language, called PLX, is designed to address three issues: (1) the program construction task faced by Automatic Programming, (2) the methods used in natural language for generating and addressing objects, and (3) the simplification of the error detection and correction task faced by a debugger.

Although not designed for any particular Automatic Programming system, XREP will be placed within a hypothetical framework in order to better focus the rest of the report. Figure 1 displays this system showing the transformation of the original input into a final program.

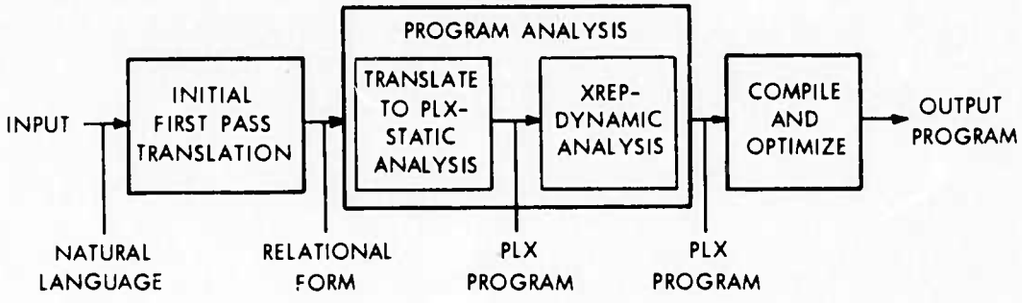


Figure 1. A hypothetical Automatic Programming system

The original input is given to a first-pass natural language translator which generates some internal form, say a relational description of that input. The next module massages that description, fixing whatever it can with its static analysis. Some of its actions might include spelling correction, reordering procedure parameters, altering colloquialisms, supplying obvious missing information, etc. The PLX program, the output from that phase, is then passed to XREP for an execution analysis. The PLX program then enters a compile and optimize phase resulting in the final product. For the moment we will assume that interaction is possible at any stage of the processing.

Within this framework all of the errors detected by XREP's debugger are English situations which may cause problems for an Automatic Programming translator. Many of these problems can be better resolved at execution time, when the dynamic context is available, than within the static environment of a translator. XREP has been designed on this principle.

The adequacy of all the internal representations, programming language included, should be measured by the success of the debugger in having compatible and understandable models of the problem and its solution. Both models are required to understand a program's behavior and have expectations of its results from which its correctness can be tested or verified. Understanding programs -- the primary focus of Automatic Programming -- can occur only in such an environment.

Generally an experimental system produces some characteristic behavior to support its claims. However, the proposed debugging methods of XREP are intended to augment the capabilities of an Automatic Programming system by providing a powerful enough framework in which to address program construction issues, as well as do debugging. They must therefore be evaluated in this larger context rather than simply as debugging facilities. Although we have obviously not built a complete Automatic Programming system as part of this effort, we will attempt later to show how the features of XREP could facilitate such a system.

1.2 PROGRAM BEHAVIOR AND EXPECTATIONS

Analyzing a program's behavior involves some expectation of its results, many of which are independent of any particular task. Halting, avoiding numeric overflow, and addressing proper data are expectations relevant to all programs, but specific expectations are obviously present as well. When they can be formally stated, the program construction task can often be automated and proved correct(ii). Unfortunately, few processes (especially long ones) can be defined so functionally. Yet informal expectations are used by human programmers to help check out their product. XREP's "intention mechanism," which is used to monitor a program's execution, is informal in the same way. The mechanism's function is to help the debugger detect flaws (i.e., deviations from expected behavior), not prove correctness. Still, the debugger can extract much information by noting what is expected and what is produced -- information certain to be useful.

Given this setting for a monitor and debugger, the next section will describe the structures and formalisms on which they work.

(ii) This research will be reviewed in Section 5.3.

1.3 *REPRESENTATIONAL REQUIREMENTS*

The choice of representations in XREP involves two design criteria: structures need to be robust enough to enable the modification algorithms to work and they need to be influenced by the thought that a user and an Automatic Programming system are considered to be the front end. The first criterion is operative, the second vague. Since no front end exists, any claims for communicability, naturalness, or closeness to real life can only be intuitive, though the claims will have strength and be backed by examples.

By dealing with existing programs and performing execution analysis on them, XREP must have a model with enough power to describe program execution, program expectations, and execution errors or anomalies. At the same time the model must "understand" the programming language involved and how it accommodates the system and the user. None of these issues can be isolated; it is their coherent organization which gives XREP the capability to perform intelligently.

XREP responds to four representational issues: data, the process description, execution behavior, and intention forms. Data refers to the naming and accessing of variables. The process description is the language for writing programs. Execution behavior is captured by a graph which depicts the progress of a running program, while the intention forms are used to measure "correctness" of that progress by monitoring its results. Discussion of the individual issues will emphasize how each influences the others.

Generic Data

The generation and addressing of data is an example of how an Automatic Programming system can accommodate the user. Typical programming languages have precise models of data. If a programmer needs to generate information, he chooses a variable name and makes the proper assignment. Thus, "X = 1", "X ← 1", or "(SETQ X 1)" are all valid examples of this action in their associated languages. This phenomenon is perhaps the best example of the difference between the formality of programming language and the informality of natural language. People do not say, "Call this pot X. Fill X with water." Programmers do exactly that. The preciseness of such an assignment makes all subsequent references unambiguous. Humans, on the other hand, use anaphoric and ambiguous references as their primary addressing mechanism. They refer to objects generically modified by a predicate: "The pot with the water," "The first person," and so forth. Often the type is replaced with a pronoun form, such as "The one with." Though these features give language its fluency, they present severe problems for the understanding of natural language.

In XREP variables are created and referenced descriptively in a manner similar to English; thus a translator need not make the rigid reference assignments to textual input. More importantly, however, by maintaining more natural constructs, a better framework can be provided in which to resolve the problems associated with flexible English references. The issue here is not to argue for the fluency and ambiguity of English, but merely to recognize that it exists and must be faced by Automatic Programming. The farther the internal translation from the original, the more difficult the communication task. Chapter 5 will amplify this claim by giving examples demonstrating the power of this representation.

The Programming Language

Due to the nature of this project, a new production language, named PLX, was designed for XREP. Because they did not need to contend with "features" of existing languages, the constructs of PLX could be designed to focus directly on relevant issues. The justification for a ***production*** language has an even deeper basis. Production languages have a simple control structure; in fact, production languages have too simple a control structure for most programmer's use, which explains not only its absence in programming shops but also its usefulness for analysis. Understanding a program's execution is simplified with a well-behaved flow of control. However, the nondeterministic behavior of some production systems, including ours, can inhibit error detection by trying fruitless backup instead of recognizing a true flaw. To aid in this case PLX has a "terminal" operator which the monitor uses in trying to identify types of failures as they arise.

Another feature of production systems is their inspectability. Since debugging is a primary concern of the system, the programs on which it operates must be easily modifiable. The production rules maintain a perspicuity which make them ideal for this task.

Finally, though not normally used in this way, production systems can impose a top-down discipline on program creation. The complexity of acquiring a problem statement from a human seems to dictate that an Automatic Programming system play an active role in the process. Top-down methods provide an excellent framework on which to base such a dialogue, with the nonterminals of the language acting as reference points for maintaining continuity. Unfortunately, traditional production systems tend to defeat the top-down benefits by having unnecessary nonterminals solely in order to produce the appropriate structure. PLX solves this problem by means of a structuring convention. The acquisition of a program is not a concern of this dissertation, but the structuring issue is.

Execution Modelling

A model for understanding a program's execution must account for many details not found in most systems. A Program Status Word or execution stack is not adequate for analyzing execution behavior. Besides presenting a current view of a process in the form of a program counter and a data base, a model must also address dynamic issues accounting for the history of the entire execution process: how control was passed and gained, when and where variables were bound, and what data are available to a particular event. At the same time, that model should reflect the program it represents. In other words, the production language and execution model should accommodate each other; the language, by simplifying the model's construction, the model, by maintaining the structure imposed by the language.

In XREP a threaded tree structure, called a Process Elaboration Graph (PEG), and an Access Graph, which is just another view of the PEG, satisfy those requirements. By emphasizing control issues and maintaining closeness to the production rules, the PEG becomes the focus of the program modification algorithms. The Access Graph emphasizes a different picture of the execution by accentuating access and scope issues. That is, given an event in the Access Graph, the data in its

scope of reference are immediately observable. The discussion of the generic data types depends upon this view of the PEG, since a graphic representation of all the bindings is necessary if complicated references are to be accurately resolved. The global nature of the Access Graph provides a natural environment in which to view structural flaws that cause access errors.

Program Intentions

Without a formal expectation of a program's performance, any debugging or understanding system will only be able to react to task-independent problems. The ability to match expected behavior to actual behavior will give a goal-oriented direction to debugging efforts. XREP accepts an ***intention string*** for exactly that reason. The string, a sequence of "observable" program events, is mirrored in PLX by a TERMINAL primitive which generates the "observable events." The system will try to match this output against the given intention. The intention string is not meant to provide a method for proof procedures; it is merely a tool to help the system produce more correct programs.

The constructs presented in this section are designed to focus on debugging and intentions, while also raising several side issues important to Automatic Programming. As representations, their adequacy can be judged only by the methods which operate on them. The next section will describe the intent and scope of those methods.

1.4 DEBUGGING EXECUTION FLAWS

The extensive preliminary discussion was intended to stress the environment concept fundamental to this report. The discussion of debugging will unify all the formalisms of XREP by providing a coherent model in which to picture the execution process. Figure 2 depicts XREP as two logical parts: a program executor and a program debugger.

A SYSTEM MONITOR is responsible for executing the PROGRAM written in the production language. The monitor records the PROGRAM BEHAVIOR in the Process Elaboration Graph, while checking that its behavior is consistent with the EXPECTATION, existing in the form of an intention string. If a difference is detected, the DEBUGGER is called. The first step is to IDENTIFY the nature of the error. That involves both CLASSIFICATION of the problem, and an EXPLANATION for its existence. Then CORRECTION can be attempted.

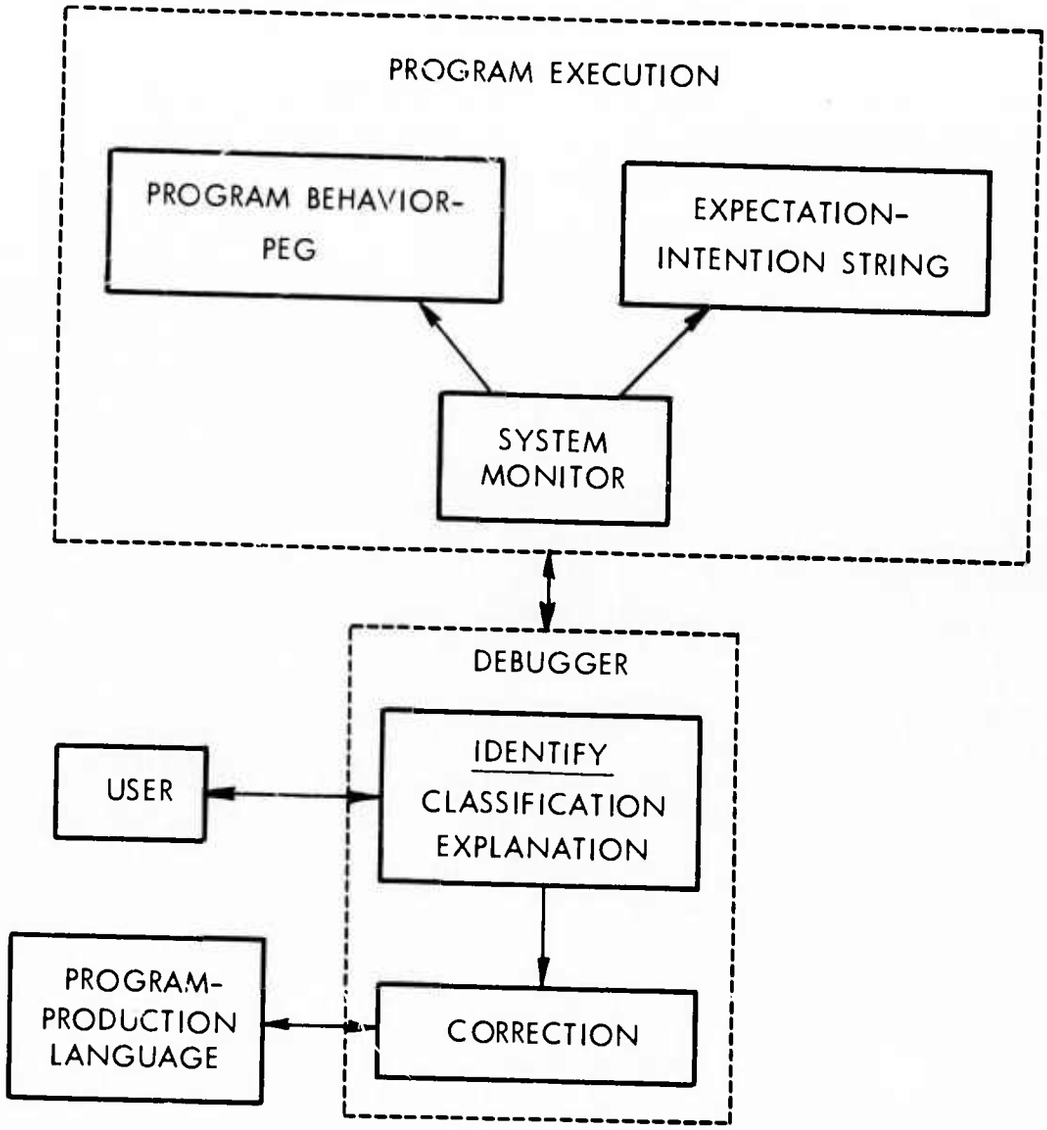


Figure 2. Model of XREP

The range of errors considered by the debugger will be limited to reflect some of the problems which can arise due to the impreciseness of English. The set of considered errors is intended to demonstrate the adequacy of XREP's representations in terms of the issues raised earlier. They should not be considered a complete or minimal set for an Automatic Programming system, but they are expected to be somewhat representative. Two are structure problems, another

is associated with loop control, and the last deals with pronomial references. All four exemplify a position taken by this system design:

> Many Automatic Programming translation problems can be resolved better by studying dynamic behavior rather than a static description.

The basis for this decision comes from the intuitive methods used to resolve the problems. All seem to be naturally suited to execution analysis.

Structural Dependencies

Computer science has become very conscious of structured programming(iii), a concept which is important mainly because it forces a programmer to design a solution to his problem carefully before coding it. The benefits of following the disciplines imposed by structured programming are numerous: the programs are easier to read, understand, and modify. These advantages are gained because of the modularity resulting from delaying various design decisions until necessary. That is, any "abstract level" of the program contributes only what it must to the overall design. English descriptions of process are notorious for doing exactly the opposite. Information is presented with no regard for the programming concept of structure. Neglecting elegance for the moment, Automatic Programming systems will have a difficult enough problem just in determining dependency issues, i.e., what data is required for a process to operate. A linear (i.e., nonhierarchic) representation of a process, coupled with a global data base, loses all the structure inherent in English. Again, the informality of English is to blame. However, the structure is there; otherwise anaphoric references, ellipsis, and ambiguity could not be viable communication tools. Automatic Programming must find that implicit structure to be able to write any program, structured or otherwise. The debugging effort for these problems accentuates the help that execution analysis can provide.

Consider an Automatic Programming system, structuring a process stated in English, trying to deal with a reference like "the next to last person." If this reference depends on the dynamic behavior of the program (as do most such references) the Automatic Programming system will have a hard time discovering, statically, whether at least two persons exist in the current context for that reference to make sense. This problem is the primary structuring issue. Assuming that the sequencing of the program is correct and two players do exist, its structure may present a context in which only one person is accessible during the "next to last person" request. In that case XREP will restructure it to make sense. The modification is made possible by the generic features of the variables and the history of the process maintained by the Process Elaboration Graph. The naming conventions imposed by typical programming languages are too rigid to facilitate this kind of analysis. In this example the problem is too much structure. A related case occurs when the "next to the last person" request succeeds, but points to the wrong person; this problem will be resolved by a similar analysis.

(iii) See [DAHL 72] for a state-of-the-art discussion of this topic.

Lack of Structure

The other structure issue appears when a program is running correctly, but is too linear; access to unneeded information typifies that flaw. That is, an event has more information than it needs to accomplish its task. The form of the generic data suggests a method of detection by having the system maintain a count of each generic type produced. If the associated maximum is known and exceeded, a problem of this type can be hypothesized. The position taken by the debugger is that an iterative process is at fault. The debugging effort involves finding the iteration point and modifying it so that the overflow is corrected.

The impetus for handling this problem comes from future execution considerations. Later accesses to this data may be ambiguous because of the extraneous data. Also, this kind of program structure is not conducive to future modifications. Since the condition is detectable, fixing it seems appropriate. Humans handle this situation by automatically maintaining contexts containing only necessary information. Unfortunately, this structure is not transmitted in their description of processes. Still, experts in various fields have the ability to effectively manage their information by structuring it to ease future access to it. The debugging effort here attempts to do the same in this special case.

Loop Control

Erroneous loop control is another "feature" of natural language. Examples will demonstrate how humans determine iteration points dynamically from imprecise and ambiguous algorithms. This problem will be viewed in terms of why it exists and how it might be resolved given the environment presented thus far. Only a partial solution will be offered, with no implementation, since a complete analysis of this kind of situation is the focus of other research projects.

Ambiguous References

Anaphoric references present the basis for the next analysis. Consider a reference like "the first one." Syntactic clues may not find the referent if the situation is ambiguous. English abounds with such constructs, forcing language translation systems to deal with them. The PLX generic data forms provide a natural interpretation for this problem as an unknown type, while the intention string gives the capability to resolve it. The ease with which this can be done strengthens the position to delay binding decisions as far as possible.

The class of errors reviewed in this section is not meant to be complete. They cover a variety of typical situations which arise when dealing with natural language. The formalisms attempt to simplify the manifestation of those errors, thus enhancing the ability to correct them. If these problems can be solved, the fluency gained in communication with humans will allow an Automatic Programming system to consider the program construction issue more directly.

1.5 PHILOSOPHY OF PROGRAM UNDERSTANDING

Since "program understanding" encompasses such a variety of efforts, a review of the intent of this dissertation is needed. Rather than formalizing requirements which define understanding, we have presented an environment in which understanding can be demonstrated. This environment includes several new constructs, whose inclusion is justified by the resulting methods. The advantage of this approach is also its weakness; the flexibility gained by having loosely connected formalisms prevents the consistency required for proof. Thus in most cases methods are heuristics, while constructs are justified in order to promote a desired behavior. Not enough is yet known about programming to impose enough structure to formalize Automatic Programming. Yet, useful results can be extracted if completeness criteria are not demanded. The environment concept occupies a middle position in a spectrum which has batch computing on one side and total Automatic Programming on the other. There is no need to sit at one end, waiting for enough progress to make the complete jump to the other. This philosophy addresses the immediate applications made possible by Automatic Programming research.

2. AN INTRODUCTORY EXAMPLE

2.1 INTENT OF THE EXAMPLE

The object of this chapter is to introduce XREP and its formalisms in the context of a particular example. An analysis of the English statement for that example will disclose communication methods unique to natural language. Section 2.3 will present a possible program for this example, emphasizing specific constructs of XREP's production language suitable for these communication methods. An execution of this program is then shown via the process graphs, followed by a general discussion of how the intention string mechanism can be applied to this problem. The goal is to show that the system constructs are both natural to English and adequate for describing and debugging processes.

2.2 THE ENGLISH STATEMENT

Backgammon, a two-person game, is the setting for the example. The rules for the beginning of the game are as follows:

> ***The game starts by having each player roll a die. The player with the largest value makes the first move.***

This example has several distinctively English characteristics which are ignored by computer languages. They will be discussed here in terms of the English, while the next section will review them in terms of their impact on XREP's production language.

The beginning of the statement, "The game starts by . . . ," introduces an immediate problem by implying that the top-level structure of BACKGAMMON is the start followed by the rest of the game, without clarifying whether the rest of the game depends on the start. Humans do not require explicit instructions to help make distinctions of this sort. However, if an Automatic Programming system is trying to structure this process and the distinction is important, any tentative decision made by its translator should be simple to undo if proven wrong. A major reconstruction effort for this common situation would be undesirable.

The example continues by introducing a structure. The game segment ". . . each player roll[s] a die" maps into the event "for every X do Y." No order is implied, while the action of each player is independent of the others. The event can be viewed in at least two levels of abstraction, as an overall action, i.e., the die rolling, or in terms of the individual players each rolling a die. The former view implies the existence of an addressable *unit* of information, whose internal composition can be retrieved only by a request which acknowledges the independent actions. For example, if the second line of the game said "The player makes the first move," an ambiguous situation arises because the player referent does not acknowledge the multiple components of the die-rolling unit. Since no individual player exists in that context, something is wrong. In order to be legal, a request to the die-rolling unit must make specific selections or refer to all the players.

Addressing this kind of multilevel structure leads into retrieval methods exemplified by the anaphoric reference in line two, "The player with the largest value." Typically, information is created by type and referenced by that type with an identifying predicate. Since the data is created in some context, enough clues usually exist to identify it uniquely. Anaphoric reference is based on this assumption; the predicate format is one way to supply characteristic information. The form of this reference is also the one necessary to address the components of the unit introduced in the previous paragraph. The predicate part of the reference offers detail which indicates knowledge of the unit's format, thus distinguishing it from a general reference like, "The player moves."

The second line of the game description points to another difference between English and programming languages in the way procedures are invoked. Programming languages use a formal parameter passing mechanism to identify a procedure and its arguments; English does not. Instead it creates information as necessary, expecting the individual procedures to search their current context and "find" what they need. In the example, a "first-move" procedure is to follow the rolling of the dice. The English description does not tell what information the first move is to use other than "the player with the largest value" precipitates that action.

The above situations are natural to English; none is perverse in any way. The design decision to model them in the production language is based on the presumption that "closeness" to English is a desirable feature for a target language within an Automatic Programming system. By accurately modelling English, the language also models English difficulties with specifying programs, thus making the manifestation of an error relatable to the original text.

2.3 A PLX PROGRAM

Figure 3 shows a program written in PLX which represents the BACKGAMMON segment introduced in the previous section.

```
BACKGAMMON := START , REST-OF-GAME
START   := (GENSEQ PLAYER -> ROLLDIE) -> COMPARE
ROLLDIE := (GENMEM DIEVAL)
COMPARE := (INSERT PLAYER.(INDEX MAX DIEVAL)) -> FIRST-MOVE
FIRST-MOVE  := (TERMINAL PLAYER.-1 'MOVES')
REST-OF-GAME := . . .
```

Figure 3. Rules for beginning of Backgammon

In order to ease the discussion, the production rules are simplified and use ". . ." in place of program segments irrelevant to the discussion(i). After the first rule is initially viewed abstractly to explain the basic operations of production systems, all the rules will be inspected from two standpoints: their role in the program and their derivation from the English.

A rule has three parts: a left-hand side, a rule separator, and a right-hand side. For the first production they are BACKGAMMON, ":=", and "START , REST-OF-GAME" respectively. The left-hand sides, also called nonterminals, represent the names of processes whose definitions are given by the corresponding right-hand sides. Thus BACKGAMMON is a process made of two parts, START and REST-OF-GAME.

To start operation, the production system finds a definition for its distinguished beginning nonterminal, in this case BACKGAMMON. Once a definition is found, it is executed. So, to play BACKGAMMON, first START is executed, then REST-OF-GAME takes control. Notice that since START is also a nonterminal, the same process that was applied to BACKGAMMON is applied to START. This recursive expansion of nonterminals stops when terminals are encountered(ii).

The symbols between the events of the right-hand sides ("," and "->") have no bearing on the order of execution; control in XREP's production language is the same as that in standard production systems. Their role will be explained shortly, during the discussion of the individual rules.

The first production

BACKGAMMON := START , REST-OF-GAME

(i) A complete definition of the language is given in Chapter 3.

(ii) See [GINSBURG 66] for a formal view of production systems (or rewrite systems as they are often called).

is a simple rule composed of two nonterminals, START and REST-OF-GAME. While the previous paragraph viewed them in terms of control, it also hinted at their top-down nature. As a description of BACKGAMMON, START and REST-OF-GAME are sufficient; they are a complete view of the game. Of course, each of these actions need be defined in terms of more primitive behavior, but notice how, as nonterminals, they could be used by an Automatic Programming system as a basis for inquiring about refinements or as a source for later questions.

Though the order of events is implicit in production rules, the scope of information for each event is not. Consider REST-OF-GAME. In standard production languages it (and its descendants) would be on a separate branch from START because both are in the same production rule. To make the information generated by START available to REST-OF-GAME, two approaches are possible: the information can be made global or some descendant of START can generate REST-OF-GAME at the right moment. Unfortunately, both solutions resolve the problem by destroying the top-down benefits alluded to earlier. Instead, XREP's language uses explicit event separators to handle this situation. If an event is to have access to its predecessor's data, then "->" is used between them; if not, "," is used. By using event separators, both dependency possibilities are retained without losing the perspicuity of the top-down description.

In the first production rule, "," is hypothesized, meaning that START and REST-OF-GAME are independent. If that assumption is wrong, just replacing "," by "->" corrects it. The simplicity of this change is meant to model the apparent absence of this problem in English, where structural dependencies seem to be determined easily and dynamically as needed. This same dynamic reconfiguration will be seen in Chapter 5, where correction of these structural flaws is expedited because of the event separators.

The second production

START := (GENSEQ PLAYER -> ROLLDIE) -> COMPARE

defines the START of the game. Corresponding to the English statement, "Each player rolls a die," the expression "(GENSEQ PLAYER -> ROLLDIE)" is the first example of a compound event, i.e., one which is not a nonterminal. The execution of this event produces a structure consisting of a series of independent actions. The "->" between PLAYER and ROLLDIE is actually superfluous in indicating that ROLLDIE is the event which originates from this instance of PLAYER. If the English had said, "A player rolls a die," then the GENSEQ (generate sequence) would be replaced by GENMEM (generate a member). Thus, both forms "each X do Y" and "an X does Y" have an interpretation in the language.

COMPARE exists as a convenience to distinguish the action of rolling the dice from the inspection of the result. Since "->" separates it from GENSEQ, COMPARE does have access to both GENSEQ and its descendants.

The third rule

ROLLDIE := (GENMEM DIEVAL)

has an example of a simplified use of the GENMEM primitive. Notice that the English did not describe

how to "roll a die," so inclusion of this rule reflects its need because of the action of the next production. In any case, this production defines the ROLLDIE process as the generation of a member of the set DIEVAL.

The fourth rule

COMPARE := (INSERT PLAYER.(INDEX MAX DIEVAL)) -> FIRST-MOVE

represents the statement "The player with the largest value moves first." The rule contains the first example of a reference to a generic data type. PLAYER.(INDEX MAX DIEVAL) follows the general form "type.exp," where "exp" is some selector function pointing to a specific instance of "type." Here, PLAYER is the type and (INDEX MAX DIEVAL) is the identifying expression.

INDEX is a function which searches the appropriate GENSEQ structure in order to find the required item -- in this case, the largest DIEVAL. Since each player is associated with a DIEVAL, pointing to a particular one identifies the PLAYER who rolled it. The effect is like saying JOHN's 5, if 5 is the largest die value.

Once the appropriate player is found, INSERT sets him up as the generator of the FIRST-MOVE event. In this sense it has the same effect as the compound event

(GENMEM PLAYER -> FIRST-MOVE)

The difference is that no new instance of PLAYER is created, an existing one is merely repositioned, anticipating future reference. So, if a descendant says "He moves ..." or "The last player mentioned," the identification of the referent will have a firm basis. This method of having procedures find their arguments is part of a heterarchical system design espoused at MIT [MINSKY 72] and examined further by [WINSTON 72]. It proposes that "smart" systems should know how to find relevant information themselves. The strict hierarchy imposed by formal parameter passing methods is not natural to English.

The next rule

FIRST-MOVE := (TERMINAL PLAYER.-1 'MOVES')

results in a terminal output event. That is, FIRST-MOVE produces a string like (JOHN MOVES), representing the end result of a process. Another example of a generic variable, PLAYER.-1, occurs within this TERMINAL event. This time the predicate refers directly to a position -- in this case, the last PLAYER mentioned. The ease of this access comes from the work of the INSERT primitive, i.e., INSERT reinstantiates a type's value, while type.-1 retrieves it.

The intention string, described later in Section 2.5, is meant to match the composition and sequence of these TERMINAL events. By placing the TERMINALs judiciously, various levels of program detail can be revealed for testing or monitoring purposes. For now, the TERMINAL represents an explicit statement of the computation's status.

This section has viewed the production language in terms of the computing capabilities necessary to write programs. The next section, in presenting the process graphs, depicts the production language as a vehicle for their construction.

2.4 *THE PROCESS GRAPHS*

The structure which maintains a record of a program's execution is called the Process Elaboration Graph (PEG). The information it contains and the form it takes were influenced by a variety of design decisions dealing with the production language, the generic data forms, and the debugging capabilities. Though reflecting all those issues internally, the PEG requires another conceptual view, called the Access Graph, to help depict the spectrum of claims made for it. By presenting a view in which the production rules maintain their original form, the PEG relates a program and its flow of control, thus becoming the focus of the debugging algorithms. The Access Graph, on the other hand, emphasizes data and scope issues by making access paths visually explicit, a feature not present in the PEG. Through the PEG is the only structure maintained by the system, the Access Graph exists to offer a more natural structure to view when access is discussed.

Assuming the players are named Joe and John, Figures 4 and 5 picture the Access Graph and the PEG for the current example. The difference between them has an intuitive basis which will be reconciled later in this section. First, the construction of these graphs will be compared to the tree produced by standard rewrite systems in order to emphasize the role of the event separators and the form of the production rules.

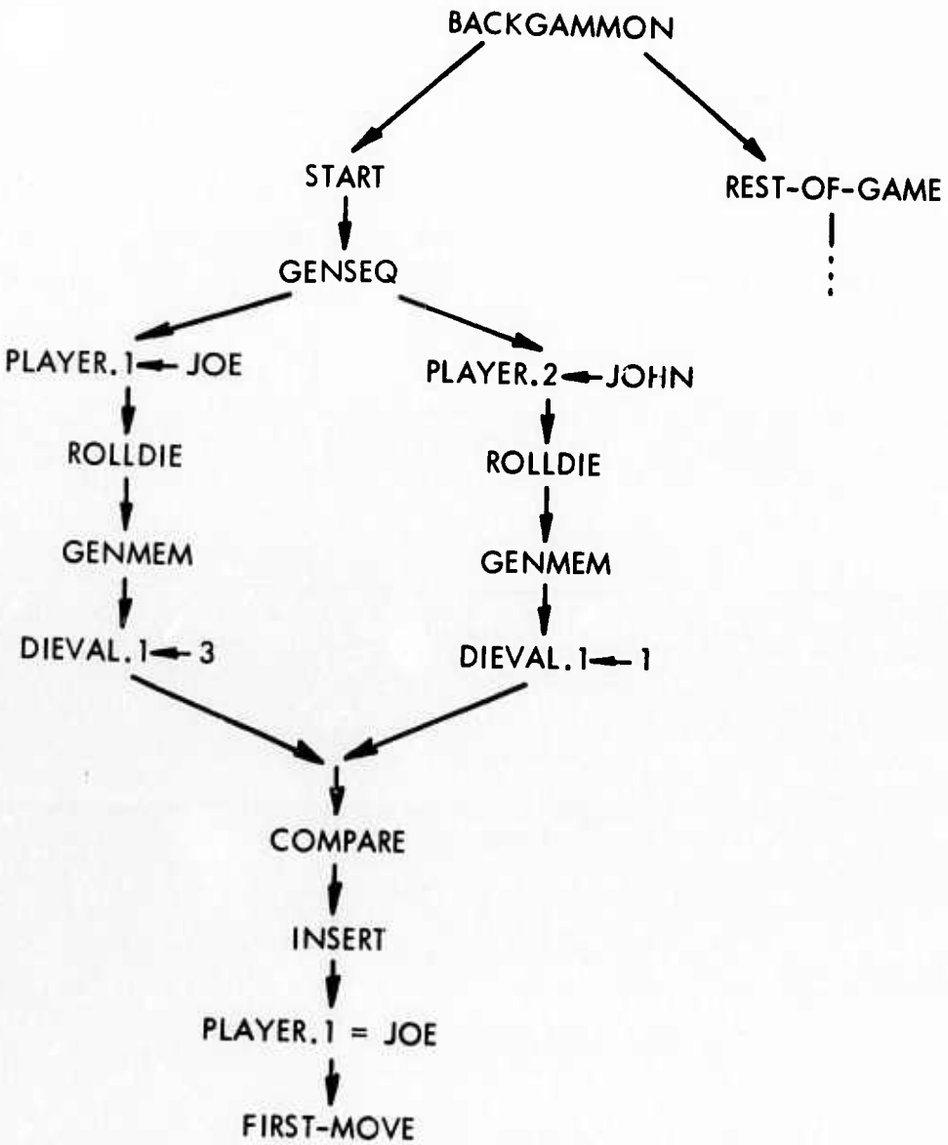


Figure 4. An Access Graph

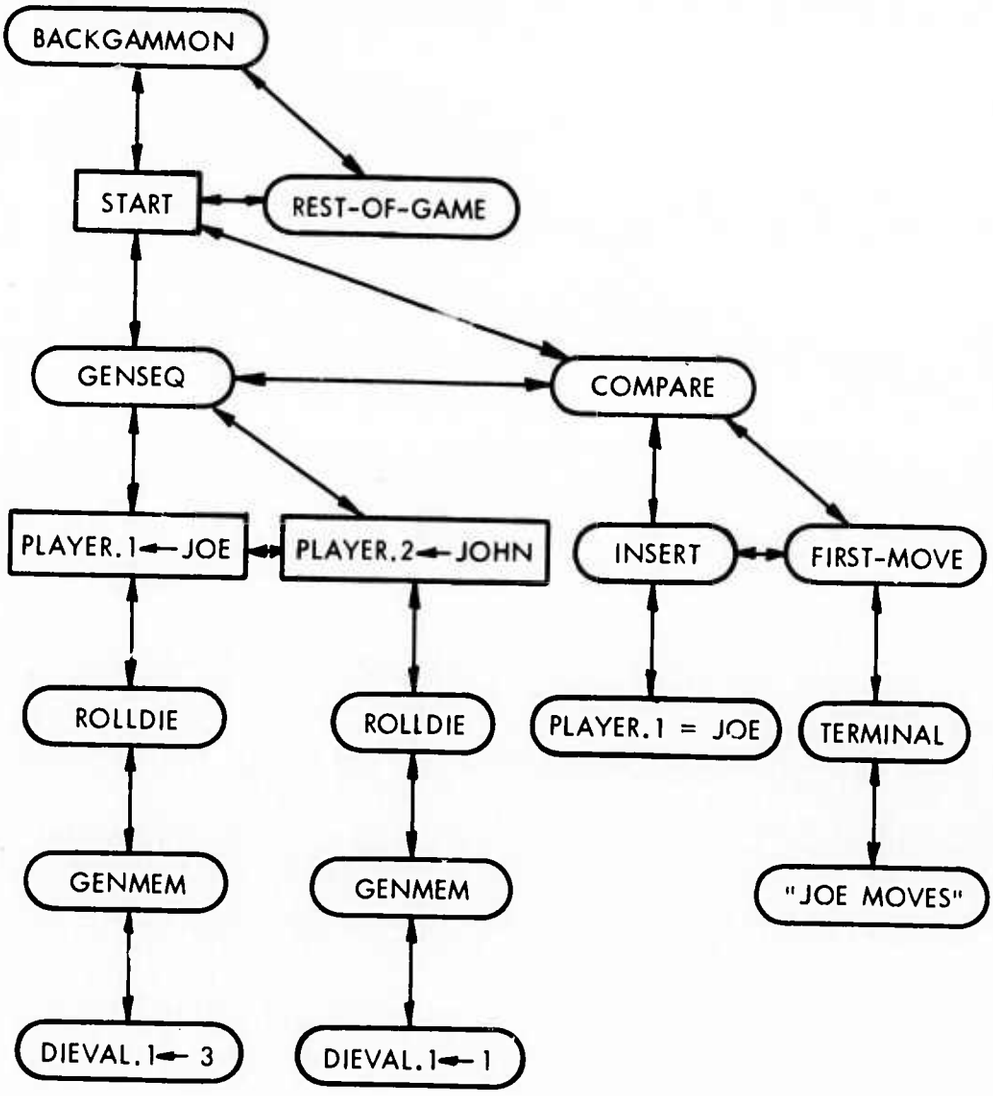


Figure 5. A Process Elaboration Graph (PEG)

XREP and Standard Production Systems

The functional events, like GENSEQ, and the event separators make XREP's production system different from others. yet by treating the functional events as nonterminals (while ignoring their semantics) and by applying one transformation based on the event separators, the rules can be made to look like those of other production systems. The transformation is as follows:

1. Whenever "A := . . . B -> C . . ." appears in a rule, change it to "A := . . . B'" and "B' := C . . . ."

2. Whenever "A := . . . B , C . . . " appears, replace it by "A := . . . B C . . . ."

Applying these transformations to the BACKGAMMON game, the rules become

```
BACKGAMMON      := START  REST-OF-GAME
START           := GENSEQ'
GENSEQ'         := COMPARE
COMPARE         := INSERT'
INSERT'         := FIRST-MOVE
FIRST-MOVE      := . . .
REST-OF-GAME    := . . .
```

A "standard" execution of this program produces the tree structure shown in Figure 6.

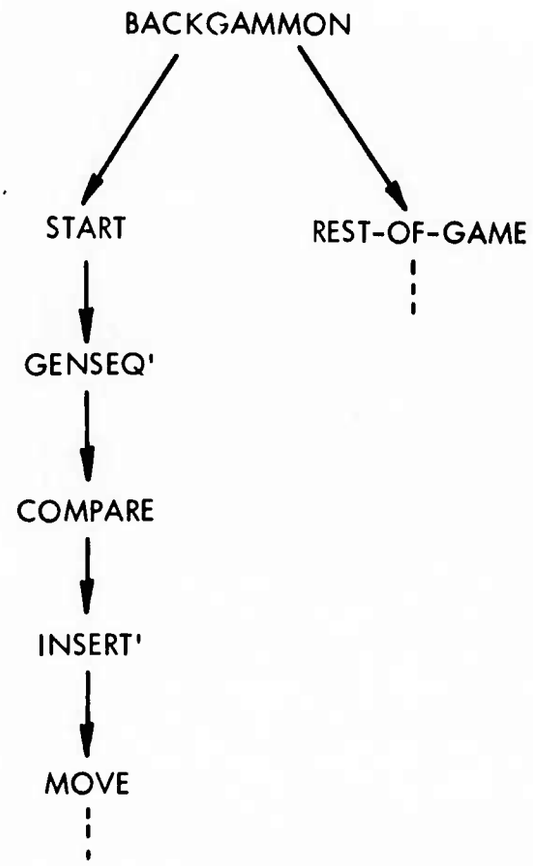
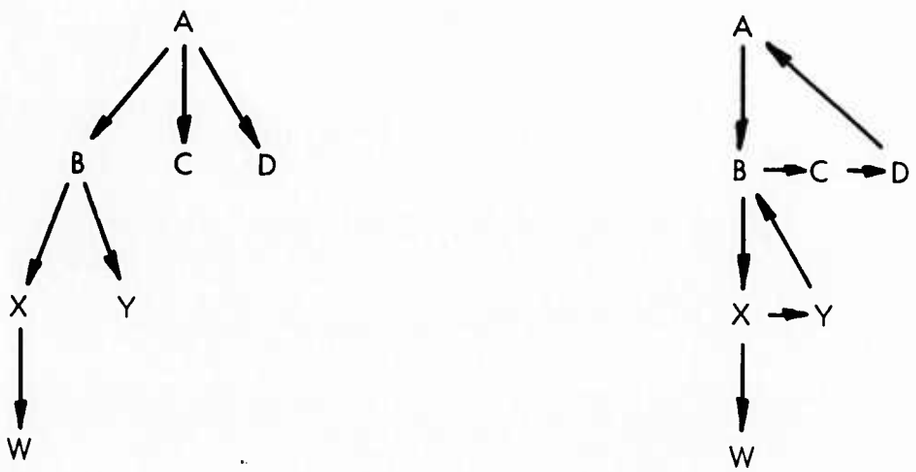


Figure 6. A standard tree structure

This traditional structure pictures an event's access path, the path from an event to the root. By containing all its direct ancestors, the access path becomes the environment in which each event carries out its task; in this sense it is like the control stack of traditional programming systems. Other than the expansion of the GENSEQ and INSERT, the tree in Figure 6 has the same structure as the Access Graph of Figure 4. Yet although Figure 6 depicts the context concept, the form of the transformed rules loses all the structural perspicuity inherent in those of Figure 3. The event

separators act as more than syntactic devices; they provide the interface which gives the production rules the structure necessary to model a process naturally.

Though the Access Graph's construction follows easily once the convention of the event separators is known, by highlighting the access issues it distorts the relation between the form of the production rules and the sequence of their execution. The PEG maintains that relationship, though at the expense of the access issues. Consider Figure 5, the PEG corresponding to the Access Graph of Figure 4. Its main feature is its closeness to the production rules. In fact, if the event separators, encoded in the shape of the events, were ignored, the PEG would represent an implementation of an n-ary tree generated from a standard production system. For example, Figure 7 shows some simple rules from a standard rewrite system and its associated tree, both in a standard and implemented form. In the standard form each father points to all his sons directly; in the implementation of this kind of tree (since that is not a convenient form) each father points only to his first son, who in turn points to his right brother, with the rightmost brother pointing back to his father.



A := B C D
B := X Y
X := W

Figure 7. Rules and trees from a standard rewrite system

The difference between the implemented n-ary tree conceptualization and that of the PEG is in the access path. In the former case the path is constructed by visiting all the father nodes, i.e., go right until an "up link" to the father is found. However, that method does not work for the PEG because of the interpretation the event separators impose on the rules. Instead every left brother is visited (hence the two-way links) and inspected to see if it is in the access path. An event's inclusion depends on its shape, rectangular if a "," follows it, or oval otherwise. Basically, a rectangular event means it is protected, an oval event means it is viewable. Thus, the method to determine the access path in an PEG, trivial to define in an Access graph, is to (1) visit the left brother, (2) if it is oval (i.e., viewable), it and all its descendants are included; if it is square, the event is protected and not part of the access path, (3) if a leftmost node is encountered, move up to the father and continue from step 1. This algorithm produces the same access path that can be read directly from the Access Graph, with the join in the Access Graph corresponding to the viewability of the GENSEQ node by COMPARE in the PEG.

The notion of access paths is crucial to understanding the duality of the Access Graph and the PEG. The collection of all access paths defines a unique tree. The PEG, under this access path mapping, thus represents one and only one Access Graph, i.e., the Access Graph is just a reconfiguration of the PEG. Both exist to focus on different aspects of execution emphasized by this report. For now this intuitive concept of access paths will suffice; Chapter 4 will detail this topic further.

Language Impact on the Process Graphs

Several language claims made in the previous section have a visual effect on the process graphs. Consider the GENSEQ structure as an addressable unit with independent branches. The access path of COMPARE contains GENSEQ and all its branches because of the viewability of the GENSEQ node in the PEG (or the simple observation that the Access Graph has the GENSEQ structure in COMPARE's direct ancestry). But an event within a particular GENSEQ branch (ROLLDIE, for example) is independent of other branches. The Access Graph merely makes each branch separate, while the PEG protects each branch from the others by making the bindings rectangular. Notice that the join in the Access Graph (after the DIEVAL's get bound) is conceptual in the PEG, reflected only by the viewability of the GENSEQ node.

The bindings, which result from generating objects, also contribute to the visual impact of the graphs. The addressing mechanisms force the spatial positioning of the data within the graph, stressing the importance of knowing not only what value an object has, but knowing when and where it got its value. Stack models also have this information (though only for a current branch of the execution), but generally make it directly available only as a debugging tool.

Another example of the language influence is in the reintroduction of the binding "PLAYER.1<-JOE" underneath the COMPARE event. By anticipating PLAYER.-1 requests, the graphs have prepared the environment for future events so that this specific information is ***where*** they expect it. This implementation supports the English which is likely to follow in the example:

"... The player with the largest value makes the first move. He ...."

Given no other information, PLAYER.-1 is a likely translation for "He." The graph is ready for that assumption by supporting the kinds of relative addresses used by English.

Review of the Process Graphs' Features

This informal discussion of the process graphs was intended to give some overall feeling for why they exist and what information each purports to carry. The following features, used to evaluate a process representation, act as a summary for this section by reiterating the main issues.

- The representation presents a dynamic view of a process.

The static view of a process is the program which represents it; the dynamic view concerns how and why different states of that program are reached. A history of the execution is required, to give not only the contextual information which is necessary to carry out the current evaluation, but also past information to help debugging in case of failures. The PEG fulfills both of these conditions.

- A spatial view of bindings is emphasized.

The flexibility of the "type.exp" data form requires that the data appear dynamically within the process representation. To separate the data from its generation point can only lead to an information loss and an unnatural (from the standpoint of English) representation of the binding process. Both the Access Graph and the PEG maintain this spatial view.

- The process representation should be easily modifiable.

The debugger will often try to reconfigure a process, sometimes drastically, sometimes trivially. In both cases the process representation should be amenable to such changes. This concern is linked to the next one:

- The process representation should mirror the corresponding program.

This condition is the main claim made for the PEG and the production language. In executing a process the manifestation of an error is often easy to detect, but assigning responsibility for that error can be difficult. An analysis of the PEG during debugging can often involve disjoint branches. Common ancestors can easily be found, often pinpointing the event responsible for the error, and thus the production rule involved. The close correspondence between the production rules and the PEG makes this information both findable and usable.

2.5 *STATING EXPECTATIONS IN PLX*

Trying to formalize methods for stating program expectations can be self-defeating; too much detail can make debugging expectations as formidable a task as debugging programs. The diversity and depth of such attempts suggest that flexibility should be the main objective. Even in trying to prove that a program is correct, the detail supplied for proof should depend on the user's judgment for rigor.

Consider the example of this chapter. The expectation for the program can depend on many things: how well the program is understood, how confident the user is of certain program components, how many checkout attempts have been made, etc. For example, all the following can be considered statements of expectations for the BACKGAMMON program:

1. (The program halts)
2. (The MOVE event is entered)
3. (JOE MOVES 3 AND 1)
4. (JOE MOVES)
5. (JOE ROLLS 3) (JOHN ROLLS 1) (JOE MOVES)
6. (GENSEQ entered) (COMPARE entered) (MOVE entered)

All are valid expectations and can be useful at various phases of the program's development. The second might correspond to a state where the start of the game is considered checked out, while the fifth represents a full observation of the external events. The point is that any level of detail should be possible for stating expectations.

The system supports this position by providing a TERMINAL primitive for this purpose. The sequential collection of TERMINAL outputs constitutes the list which must match the intention string. In Figure 3 the rule

```
FIRST-MOVE := (TERMINAL PLAYER.-1 'MOVES')
```

has this TERMINAL event. An intention string for this program segment would therefore be (JOE MOVES). If the intention string is to be (JOE ROLLS 3) (JOHN ROLLS 1) (JOE MOVES), then the rule

```
ROLLDIE := (GENMEM DIEVAL)
```

could be changed to

```
ROLLDIE :=  (GENMEM DIEVAL) ->
            (TERMINAL PLAYER.-1 'ROLLS' DIEVAL.-1)
```

Again, the level of detail depends upon the placement of the TERMINAL events.

By treating expectations this way, XREP can be used as a parser; the intention string is the input, the TERMINALs guide the parsing. The production system is, of course, a perfect vehicle for carrying out this analysis; many production systems are used in some kind of parsing operation. The nondeterministic behavior of a production system finds a successful path though the rules, while masking false attempts.

Another observation about the intention string mechanism keeps it in proper perspective. The ability to state program expectations is a tool to aid in verifying programs, yet the intention string has a "test-case" flavor with little formal basis. As a result, when a program matches a particular string, little more can be said other than the program matched that particular intention string. While certainly no basis for proof, a successful parsing does have some measure of correctness to it. Dijkstra said that this process "can be used to show the presence of bugs, but never to show their

absence!"(iii) Though true, that statement does not reflect how useful that detection can be in debugging errors. Formal proof methods give few indications as to the cause of a failure when one is detected. As will be shown, the intention string mechanism provides a good environment for detecting and correcting bugs.

2.6 *SUMMARY*

By analyzing an English example from an Automatic Programming viewpoint, several situations unique to natural language and traditionally ignored by computer systems have been uncovered. Not only are they a basis for XREP's language and constructs, but they also represent the focus for the debugging algorithms. The Automatic Programming paradigm offers both a new framework in which to address representation issues and new criteria by which to judge their adequacy. This dissertation points to human communication methods as a source for its language representations, while claiming that close modelling of natural language problems plus the ability to resolve them measure the representations' adequacy and worth. The hypothetical role of Automatic Programming makes XREP theoretical and open-ended; as a test-bed for representational ideas, XREP can ignore the severe problems associated with producing a "closed" system. The results of succeeding chapters should be viewed in this light.

(iii) O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, ***Structured Programming***, ed. by C.A.R. Hoare (New York:Academic Press), 1972, page 6.

3. THE PRODUCTION LANGUAGE - PLX

3.1 PRODUCTION SYSTEMS IN GENERAL

The derivation of a computer program from an English statement by Automatic Programming systems requires a diversity of expertise which starts with understanding and representing natural language and concludes with debugging and proving the correctness of the generated program.

If this process takes place in an interactive environment, then both the Automatic Programming system and the user must have a target to provide direction to any discussion. That target is the program being generated; thus the programming language is the vehicle of the process acquisition dialogue.

The importance of our programming language, PLX, thus rests on its central role as the language in which the user's program (after translation) is stated and the system's understanding is based. It is therefore natural that it be designed to reflect the handling of variables and structural properties of natural language in a maintainable manner, and to address the computing issues which arise in understanding programs: what computer resources are needed during execution, how execution should be recorded, and how this recording can be used to debug the program (within the user's framework).

Although our main concern in this report is with the automatic derivation of computer programs from natural language statements, we must first develop a detailed understanding of how these programs will be represented and how, as we will show, this aids in the automatic derivation process. The worth of PLX thus depends not on how well it compares to other programming languages, but how well it responds to the expressiveness of English and the functionality questions faced by the Automatic Programming generator and debugger. The first step in our evaluation of PLX concerns the decision to make it a production language.

Psychological Considerations

The acquisition of knowledge from human protocol presents psychological considerations in designing the appropriate model for that information. Justification of PLX as a production system on such grounds can only be hypothesized by investigating other works whose primary task was the actual use of such protocols.

THE PRODUCTION LANGUAGE - PLX

In their study of human problem solving, Newell and Simon theorize that the actual organization of human programs closely resembles a production system organization. As they claim,

> A production system encodes homogeneously the information that instructs the information processing system how to behave.(i)

That is, no division exists between program information and program control flow except that order of productions may carry additional information. Further,

> In a production system, each production is independent of the others -- a fragment of potential behavior. Thus the laws of composition of production systems is very simple: manufacture a new production and add it to the set.(ii)

Information from humans does not come in a concise algorithmic form. Instead, it seems to be fragmented into related segments. This style of input is well suited for production rules.

> The productions themselves seem to represent meaningful components of the total problem solving process and not just odd fragments.(iii)

Though information is fragmented, each piece in the form of a production rule identifies a coherent idea presumably complete unto itself. If this concept is used as a goal, an Automatic Programming system can use a production rule as a subgoal in trying to produce a whole program.

The theory which embodies those claims becomes the basis for the protocol analysis done by Waterman and Newell in their PAS-II system [WATERMAN 73]. In it, information for analyzing protocols is represented as production rules in a pattern-action format. The application of those rules to nodes of their Problem Behavior Graph models the acquisition of a new piece of knowledge derived by the subject. The psychological framework of their work makes their use of production systems relevant beyond the mere computing capabilities inherent in such a system.

The DENDRAL work ([BUCHANAN 69], [BUCHANAN 71], [BUCHANAN 72], among others) is another study which uses a production system to model one part of its knowledge base. DENDRAL tries to find chemical structures from mass spectrometer data by using heuristics which were originally hand coded from dialogues with chemistry experts. Later, these ideas were encoded within a production system as situation-action rules in order to separate the theory from the program. The flexibility gained is used to the authors' advantage.

(i) A. Newell and H.A. Simon, ***Human Problem Solving***. New Jersey: Prentice-Hall, 1972, pg. 804.

(ii) Ibid.

(iii) Ibid.

> Changing the theory, then, involves little actual reprogramming. This allows experiments to be carried out with different versions of the theory, a very useful feature when dealing with a subject as uncodified as mass spectrometry.(iv)

The Automatic Programming problem is similar to the protocol analysis of PAS-II and the DENDRAL program in the kind of original data it starts with and the nature of the finished product, i.e., the input is the result of a process whose structure must thereby be inferred. By dealing with natural language, Automatic Programming must also contend with the fragmented nature of human dialogue, reinforcing XREP's decision to use a production language for dialogue modelling. Thus, the above quotation is also applicable to Automatic Programming.

Functional Considerations

The psychological hypothesis of the last subsection was meant to give some intuitive basis for the production language of XREP. The functional considerations have direct impact on this report, since many of the results will claim that the character of the productions made the appropriate analysis possible.

Functionality, however, does not refer to the computing power of the language. Since most languages have Turing machine capabilities, the range of computable functions is the same. Instead, functionality should be measured by the auxiliary benefits gained by a particular notation. In XREP, the debugging capabilities are often made feasible because the program is a series of "independent" production rules, each analyzable as an entity. By isolating a program segment this way, a specific analysis has a clear domain on which to work. Several works have been successful because of this property of production systems.

In his program for automating the learning of heuristics, Waterman used production rules for representing them, claiming that this representation technique "permits separation of the heuristics from the program proper, provides clear identification of the individual heuristics, and is compatible with generalization schemes."(v) He was intent on making the theory, represented by the production rules, an entity amenable to analysis.

The PAS-II system mentioned in the previous subsection is the vehicle for Waterman's latest work, ***Adaptive Production Systems***, in which production language programs are modified automatically by having the system generate and insert new production rules as learning takes place [WATERMAN 74]. Again, the nature of production systems aids his analyses.

(iv) B.G. Buchanan, G.L. Sutherland, and E.A. Feigenbaum, "Rediscovering some Problems of Artificial Intelligence in the Context of Organic Chemistry." ***Machine Intelligence 5***. Ed. by B. Meltzer and D. Michie (Edinburg: Edinburgh University Press), 1969, pg. 274.

(v) D.A. Waterman, "Generalization Learning Techniques for Automating the Learning of Heuristics." ***Artificial Intelligence***, Vol. 1 (1970), pp. 151.

A different study used a production system to represent inference rules for natural language relations [LINGARD 72]. Lingard and Wilczynski used a Backus Normal Form (BNF) representation for stating the interaction between relations. Thus a rule like "GF -> F F" could represent the fact that a grandfather (GF) is the father (F) of the father. Their system could accept requests like (JOHN GF JOE) and deduce its truth by using the grandfather rule, and the two assertions (JOHN F FRED) and (FRED F JOE). By representing the relation interactions this way, a uniform parsing algorithm could be used to carry out the analysis within an associative data base. In his Ph.D. dissertation Lingard continues that investigation [LINGARD 75].

The inspectability and accessibility of production rules are the main issues of this discussion. If XREP's debugger is to function effectively, it must work on a representation that is responsive to the requests which might be made of it. In Chapter 5, the scope of information needed by the debugger will emphasize these points.

Other functional benefits of the production language will be discussed later in Chapter 5 and Section 3.5 when enough of XREP has been detailed to adequately state the claim. The rest of the section is devoted to PLX, its environment, terminology, and primitives.

3.2 THE PROGRAMMING ENVIRONMENT FOR PLX

The production language character of PLX comes strictly from its control flow behavior. The design of its other facilities was influenced more by the environment in which PLX was programmed than by classical production language issues. To put the capabilities of PLX's primitives in the proper perspective, that environment will be described first.

XREP is written in INTERLISP using the data-base extensions of the AP1 language [BALZER 74a]. AP1, a LISP-based pattern match-language of the PLANNER(vi) generation, is tailored for the Automatic Programming project at the USC Information Sciences Institute. The properties and peculiarities of AP1 will not be detailed here; only the facilities borrowed from it will be considered.

The data base is associative; information is stored as tuples whose first item is the relation which associates the others, in either a positional or keyword manner. Any item of a tuple, including the relation, can itself be a tuple. Neglecting the question of variables and literals for the moment, all the following are legitimate entries:

```
(FATHER FRED BOB)

(BETWEEN BOTTLE (CHAIR TABLE))

(PARAMETER ROUTINE A B (C D))

((COMPOUND RELATION) X Y)

(KEYRELATION (KEYWORD1 X) (KEYWORD2 Y))

(KEYRELATION (KEYWORD2 Y) (KEYWORD1 X))
```

(vi) See [BOBROW 74] for a review of this generation of AI languages

The last two are equivalent examples of keyword tuples. The ambiguity of which type of relation is which (since they all look the same) is resolved by forcing each relation to fall into disjoint classes, either positional, keyword, or function (described below). So, in the examples, if KEYRELATION is declared keyword, the last two tuples are the same. If KEYRELATION is positional, then they are, of course, different.

Each tuple is assigned a unique name and stored in a named context given in its assertion. These contexts can be hierarchically organized for retrieval purposes and are under user control. The contexts effectively segment the data base into isolated sections, while the context hierarchy joins the sections as the user wishes.

Another important feature of AP1 comes from allowing its predicates and patterns to consist of an arbitrary mix of LISP functions and AP1 expressions. For example, FS* is an AP1 function whose form is

(FS* <variable> <pattern>)

This function matches the pattern, but returns the value of the variable mentioned. In this sense, FS* acts as a selector function based on the variable in the pattern. Thus

(FS* NUMBER (AGE NUMBER BOB))

says to find a NUMBER such that NUMBER is the AGE of BOB. If the retrieval is successful, NUMBER is bound to the desired value, which is then returned as the value of the FS* expression. If the retrieval fails, the returned value is NIL, the false atom of LISP.

Another possible expression is

(FS* NUMBER (AND [WIDTH NUMBER BOARD][GT NUMBER 10]))

whose interpretation is to find a NUMBER larger than 10, which is also the WIDTH of a BOARD. This example shows a mixing of an AP1 expression, (FS* . . .), two LISP predicates, (GT . . .) and (AND . . .), and an AP1 tuple, (WIDTH . . .). This marriage permits a great deal of power and convenience by allowing the user the expressiveness of both systems without restricting him to either.

3.3 *PRELIMINARY TERMINOLOGY*

The following terminology appears throughout the description of PLX. Though some of the terms have been used before in a loose manner, they will now be linked more closely to the production language.

- An *event* is either a simple or a compound event.

- A *simple event* is an atomic element to be used as a nonterminal.

- A *compound event* is either (1) a parenthesized expression whose first element is a system primitive or (2) an expression with events separated by "," or "->".

- A *node* is either a simple event, a type 2 compound event, or the result of executing a type 1 compound event. It has the following properties: (1) it can only have one ancestor and (2) all generated offspring must be new nodes (hence no loops).

- A *typed variable* is a type together with an identifying expression (separated from the type by a "."). The expression can be either a generation number or a function which points to a particular binding -- for example, PLAYER.1, and PLAYER.(INDEX MAX DIEVAL).

- A *generation number* is an integer which identifies the relative position of a variable type in a particular path from a point in the generation tree.

Thus PLAYER.2 defines the second player mentioned from some point, while, by convention, PLAYER.-1 refers to the last player inserted into the PEG, PLAYER.-2 to the next to last player, and so forth.

- An *access path* from an event in the tree is the unique ancestor chain of events and nodes up to the root.

In defining what an event can reference during its execution, the access path becomes the environment for any evaluation done by the event.

3.4 PLX PRIMITIVES

The current version of PLX has six primitives whose syntax and functional behavior will be given here in an informal manner. The next section will give a formal description of each primitive showing its effect on the PEG, while at the same time describing how the event separators cause the primitives to interact.

The form of a production rule is:

<parent-def-name> := <event> [{ , | -> } <event>]*

In other words, a valid rule is one whose right-hand side is one or more events separated by "," or "->". Besides simple events (i.e., nonterminals), an event can take on any of the following forms:

(GENMEM type AP1-predicate next-event)

(GENSEQ type AP1-predicate next-event)

(COND AP1-predicate)

(INSERT type.AP1-expression)

(TERMINAL AP1-expression)

(FUNCTION AP1-expression)

The GENMEM event given by

(GENMEM type AP1-predicate next-event)

binds a local variable, making it the "generation" point for "next-event." The value of the variable is chosen from the global data base by the AP1 request

```
(LOCAL (ENTITY)
       (MATCH (AND  (AMO ENTITY type)
                    AP1-predicate )))
```

LOCAL is an AP1 function which creates local variables -- in this case only ENTITY. MATCH is another AP1 function which tries to match the pattern given -- in this case (AND (AMO ENTITY type) AP1-predicate). The pattern's interpretation is to find an ENTITY such that ENTITY is a member of (AMO) the set "type" while also satisfying the AP1-predicate. The presence of the AP1-predicate, ignored in the example of Chapter 2, acts as a filter between the data base and the potential values. So, for example, if the data base has the following assertions:

```
(AMO 1 DIEVAL)
(AMO 2 DIEVAL)
(AMO 3 DIEVAL)
(AMO 4 DIEVAL)
(AMO 5 DIEVAL)
(AMO 6 DIEVAL)
```

then

(GENMEM DIEVAL T (-> NEXT))

will pick any of the DIEVALs, while

(GENMEM DIEVAL (EVENP ENTITY) (-> NEXT))

will consider only the values 2, 4, and 6, since EVENP is a LISP predicate which tests for the "evenness" of a number. In either case an appropriate DIEVAL is chosen and assigned to DIEVAL.1 if this is the first DIEVAL to be bound, DIEVAL.2 if this is the second, and so on. Execution of NEXT follows this binding process.

The effect of the GENMEM statement is to produce a variable which is ***local*** to the current path of the program. In many production systems, all actions depend on a global data base; there is no notion of local variables. In PLX, the typed variables, as generators for future events, act as locals, a feature which gives XREP the capability to contend with questions about data structuring.

Once the binding takes place, "next-event" is executed. If some failure occurs later, backtracking may return processing to the GENMEM for selection of a different value, making GENMEM a "choice point" in the execution of a program.

The GENSEQ event given by

(GENSEQ type AP1-predicate next-event)

has the same action as a GENMEM event, except that all values of "type" which satisfy the AP1-predicate are chosen, each of which is to be followed by "next-event." The effect is like having n independent (i.e., no interaction) GENMEM events, where n is the number of values which pass the AP1-predicate. The GENSEQ is not meant to model a loop, but instead models a structure of disjoint actions which would otherwise be difficult to represent.

(COND AP1-pred) is a predicate event which acts as a filter to the current production rule. When a COND event is encountered, it is evaluated. If its result is NON-NIL, the processing proceeds normally. If it results in NIL, then a FAILURE is detected and processing backs up to the last choice point: a GENMEM or a rule choice (to be explained in page 43).

If COND is the first event on the right-hand side of a production, the effect is very close to the situation-action pairs of the production systems found in DENDRAL and PAS-II, or the pattern-invoked procedures of PLANNER. That is, a rule is chosen and acted upon if the situation (COND) matches. The generality of AP1-predicates gives the COND event arbitrary testing power.

(INSERT type.exp) is an event used to "find" a specific typed variable bound in a preceding event and to reinsert it into the local context. A GENSEQ or GENMEM must be an ancestor of the INSERT and the search for "type.exp" must be successful. The expression "exo" is arbitrary and

must have a valid interpretation, i.e., it must point to a specific bound instance of "type." If no PLAYER has been bound in either a GENSEQ or GENMEM, then

(INSERT PLAYER.<anything>)

is erroneous. The effect of the INSERT is to reinsert the typed variable into the PEG (without giving it a new generation number) for future references.

(TERMINAL API-exp), by evaluating API-exp and "outputting" the result, acts as the program's interface to the outside world. If XREP is in a monitor mode, then the collection of TERMINAL event computations, in the order of their occurrence, must match the given intention string.

(FUNCTION API-exp) evaluates API-exp for its effect only. Since the control structure of PLX includes automatic backtracking for certain failures, the effects of FUNCTION may need to be undone. However, due to the anticipated frequency of FUNCTION events, state saving prior to execution may be impractical. The solution involves the use of API contexts and a policy decision. Each FUNCTION statement is given a new API context, linked hierarchically to existing ones, in which to make any new assertions that affect the state of the world. If this event is then to be eliminated by backtracking, then XREP needs only to remove its context from the hierarchy to undo all its effects. As long as the event has not changed any globals, its removal will be clean.

3.5 *FORMAL DESCRIPTION OF PLX*

A formal description of PLX will be given by first viewing abstract productions and the evaluation environment created by the event separators, next reviewing the control flow of the production language, and then showing how each primitive maps into the PEG. When the semantics of PLX are defined in terms of the PEG, the description of the language becomes operational, giving a firm interpretation to any construct while also making any structural changes to the PEG during debugging immediately relatable to the language.

Abstract Productions and Event Separators

The right-hand side of a production was shown to be a sequence of events with event separators, either "," or "->", between each pair. The event separators affect the evaluation environment of any event, a concept to be detailed in Chapter 4. Now they will be analyzed for their impact on the PEG and Access Graph only.

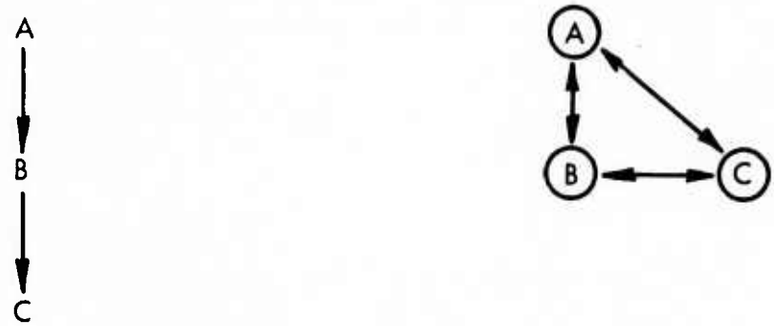
Figures 8 and 9 show the simplest rules involving an event separator. In Figure 8 the "," between B and C means that B is protected from C, reflected in the Access Graph by having B and C

on separate branches from A, and in the PEG by making B a rectangular event(vii). The evaluation environment for an event consists of the global data base plus all the information in its access path. In the Access Graph an event's access path is obvious, consisting of all events "above" it. In the PEG the access path is not so clear, since each right-hand side produces a single level under its father; the structure explicit in the Access Grapn is implicit in the PEG. Chapter 4 will show how to derive access paths from the PEG. For the purposes of this chapter, look at the Access Graph when this information is necessary.

In Figure 9, B is viewable to C, because "->" separates the events. Thus C is under B in the Access Graph, and B is oval in the PEG. This configuration means that C has access to everything generated by B, a situation which is obscured in the Access Graph, since it looks as if B has already done its work by generating C. However, the PEG clarifies this misconception by showing that B can still generate information, since it is currently an unopened leaf of the tree.

Access Graph PEG

Figure 8. A := B , C



Access Graph PEG

Figure 9. A := B -> C

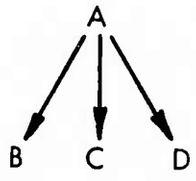
(vii) Since the rightmost event in the PEG has no "brother" successor, its shape is immaterial.

Figures 10 through 13 show all the possible productions with three events in the right-hand side of the rule. Again notice in the PEG that one production rule results in one level under the father. The PEG construction for a production is trivial; write down all the events, if "," follows one, make it rectangular, otherwise make it oval (this accounts for the fact that the last event is always oval, since no event separator follows it). The construction of the Access Graph is not so obvious, though still not difficult. The algorithm is as follows:

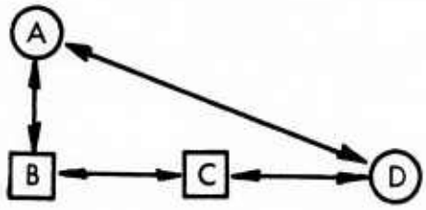
1. Write the first member of the right-hand side under the left-hand side nonterminal.

2. For each successive (event-separator event) pair, if the event separator is ",", then write the event down as a new branch under its predecessor's father; if the event separator is "->", then write the event under its predecessor.

For example, in Figure 12, B is written under A according to step 1. Next the pairs (-> C) and (, D) are considered in order as stated in step 2. Since "->" precedes C, C is written under B. Then, since "," precedes D, D is written as a new branch under the father (B) of its predecessor (C), resulting in the desired tree.



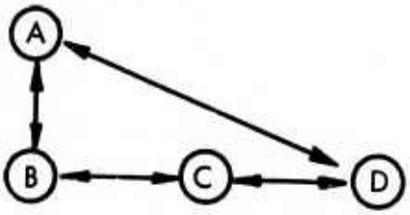
Access Graph



PEG

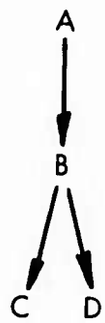
Figure 10. A := B , C , D

Access Graph

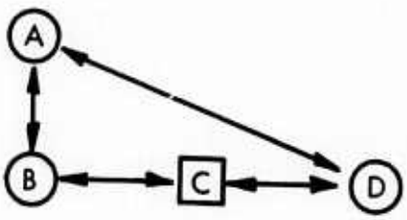


PEG

Figure 11. A := B -> C -> D

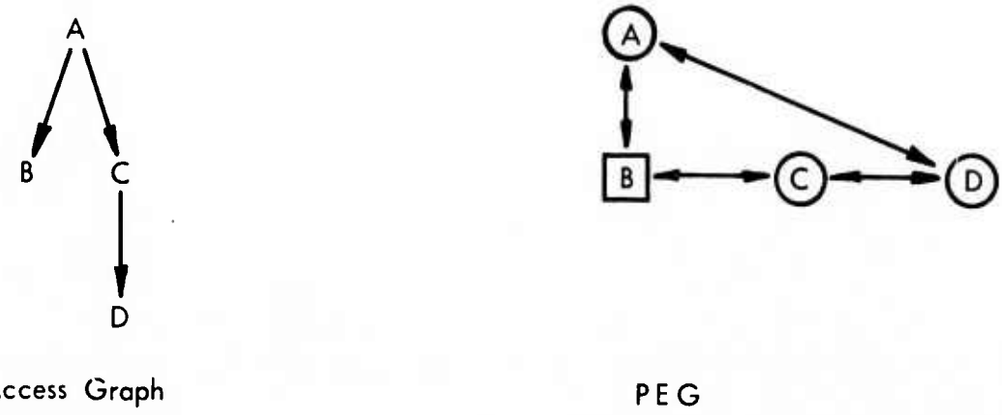


Access Graph



PEG

Figure 12. A := B -> C , D



Access Graph PEG

Figure 13. A := B , C -> D

Figures 14 and 15 picture two Access Graphs still unaccounted for. Conceptually, they can be thought of as representing the production rules given in their associated figure. However, no configuration of the PEG can account for the parenthesized expressions (B -> C) or (B , C), called a type 2 compound event in Section 3.3, while still maintaining the conventions that each production adds just one level to the PEG. The problem is fortunately not important and is circumvented by forcing rules like

A := (B -> C) , D

to be rewritten as the pair

A := temp , D

temp := B -> C

This transformation has no substantive effect other than to add an extra nonterminal in the Access Graph and introduce another level in the PEG. For this reason type 2 compound events will not be considered further.

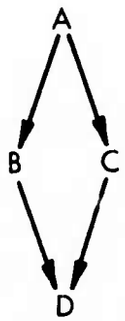


Figure 14. A := (B , C) -> D

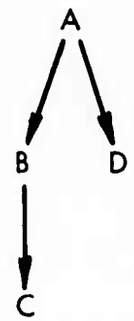


Figure 15. A := (B -> C) , D

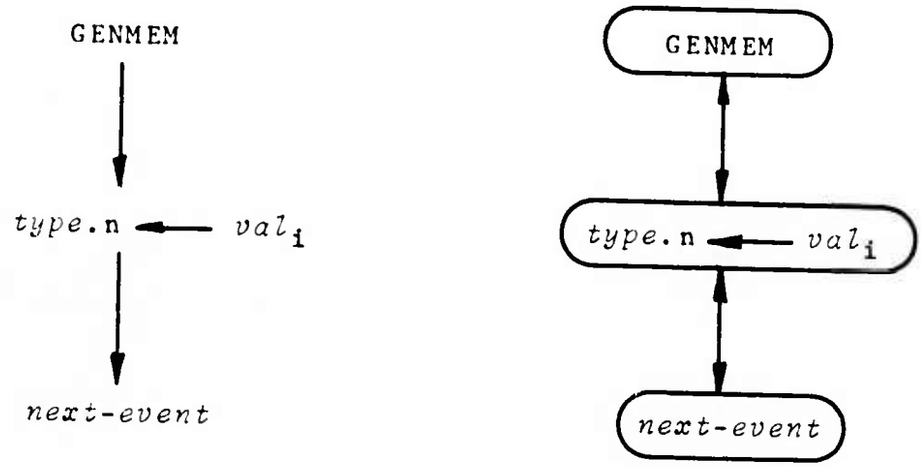


Figure 16. Graph structure of a GENMEM event

PEG Mapping of PLX's Primitives

The GENMEM event given by

(GENMEM type AP1-pred next-event)

produces the structure shown in Figure 16. The generation number n assumes that n-1 occurrences of "type" exists in the access path of this GENMEM. A member of the set "type," "vali" satisfies the AP1-pred. If no type is found, then this event fails, leading to backtrack. If a GENMEM is backed on to, a new value of "type" is picked.

The GENSEQ event is given by

(GENSEQ type AP1-pred next-event)

results in the structure of Figure 17. The generation numbers start at n, as in the GENMEM event, and end at n+m-1, where m is the number of the vali which satisfy the AP1-pred. The bindings are rectangular, since each branch is to be independent of one another -- a situation visually apparent in the Access Graph.

The INSERT primitive given by

(INSERT type.AP1-exp)

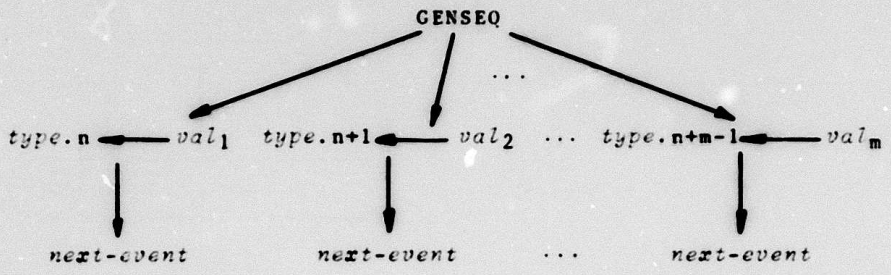
has the simple structure of Figure 18. The form "type.number=value" reflects the generation number and the value of the found "type." If type.AP1-exp does not point to a unique binding, this statement fails and backup takes place.

The TERMINAL primitive given by

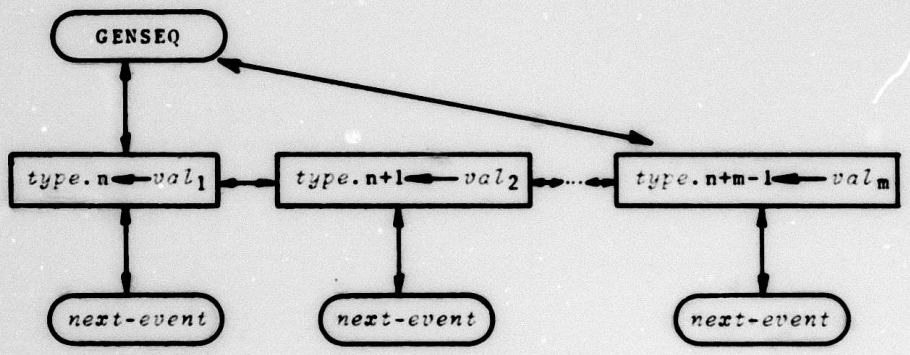
(TERMINAL AP1-exp)

is seen in Figure 19. The "result" of evaluating AP1-exp is inserted into the PEG for future reference.

The other two primitives, FUNCTION and COND, add nothing to the PEG other than their name, since they exist for their immediate effect only.



Access Graph



PEG

Figure 17. Graph structure for a GENSEQ event

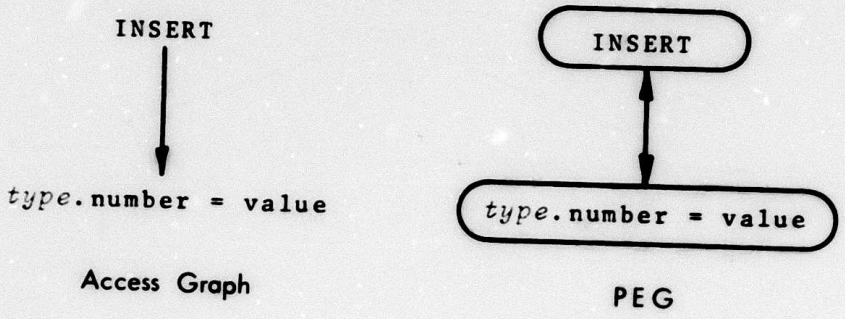


Figure 18. Graph structure for an INSERT event

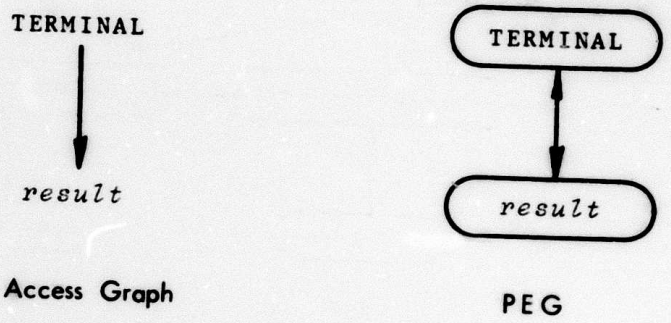


Figure 19. Graph structure for a TERMINAL event

Control Flow in PLX

The control flow in PLX depends upon the state of the PEG. Both will be described by tracing the execution of the program given in Chapter 2. First, a short overview of the processing procedure will be given.

At any point during execution of a program, XREP focuses on the "leftmost" unopened event in the PEG, called the CURRENT-EVENT. "Unopened" means that the event is a nonterminal which has not yet been expanded or a PLX primitive which has not been executed. In the nonterminal case, XREP chooses an applicable production from the set which has CURRENT-EVENT as its left-hand side. If the set has more than one possibility in it, then a "choice point" is set up for backtrack purposes if the chosen rule leads to failure. Thus XREP maintains two kinds of choice points, one in picking rules for nonterminals, the other for picking types in GENMEM events.

Once an event has done its work, control passes to the next-leftmost unopened event, determined by an algorithm called SUPER-NEXT. The algorithm is as follows:

a) If CURRENT-EVENT has a downward pointer, take it and go to step b. Otherwise go to step c.

b) If the event is unopened, make it the CURRENT-EVENT. Otherwise go to step a.

c) If the event has a right pointer, take it and go to step b. Otherwise go to step d.

d) Take the upward pointer (which must exist or step c would not have failed) and go to step c.

Basically, SUPER-NEXT is a downward tree search for the first unopened event. Step a moves down the PEG, step b tests, while steps c and d move up and to the right. The BACKGAMMON program will now be traced (the program is repeated for convenience with syntactic updates).

```
BACKGAMMON := START , REST-OF-GAME
START    := (GENSEQ PLAYER T (-> ROLLDIE)) -> COMPARE
ROLLDIE := (GENMEM DIEVAL T)
COMPARE := (INSERT PLAYER.(INDEX MAX DIEVAL)) -> FIRST-MOVE
FIRST-MOVE  := (TERMINAL PLAYER.-1 'MOVES')
REST-OF-GAME := . . .
```

Figure 3. Rules for beginning of Backgammon

The program starts with BACKGAMMON as the CURRENT-EVENT.

1. Since BACKGAMMON is a nonterminal, a rule is chosen and attached to the PEG. For this nonterminal the rule is:

BACKGAMMON := START , REST-OF-GAME

Figure 20 shows the current PEG. Notice that START is rectangular due to the "," which follows it in the production.

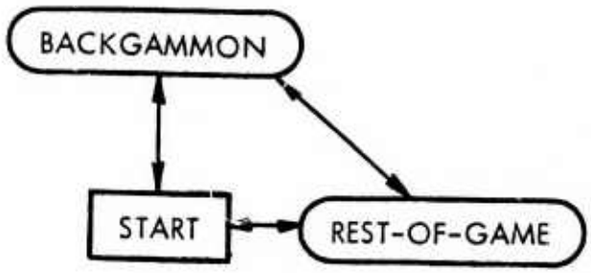


Figure 20. PEG after step 1

Application of SUPER-NEXT to BACKGAMMON makes START the next CURRENT-EVENT.

2. Since START is a nonterminal, a rule is chosen for it and attached as in step 1 above. The rule

START := (GENSEQ PLAYER T (-> ROLLDIE)) -> COMPARE

results in Figure 21 with control passing to GENSEQ.

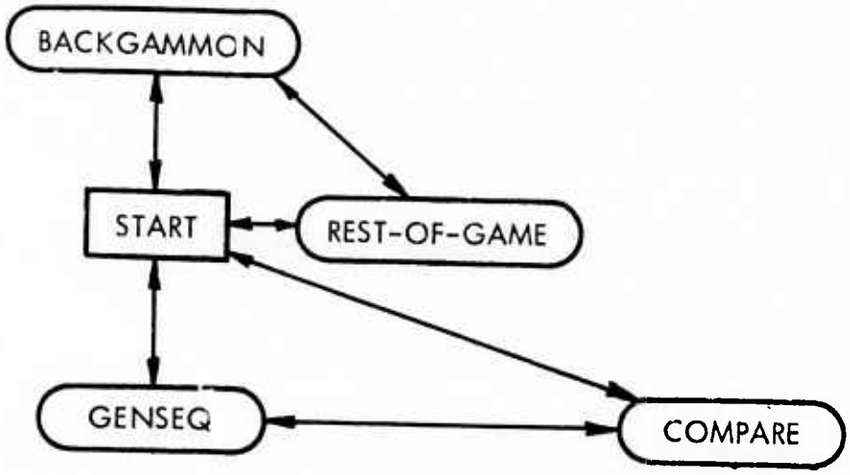


Figure 21. PEG after step 2

3. Since GENSEQ is one of the system primitives, it is executed producing the PEG in Figure 22. The two generation numbers for PLAYER are 1 and 2 because this is the first instance of this type. Control now passes to ROLLDIE under PLAYER.1.

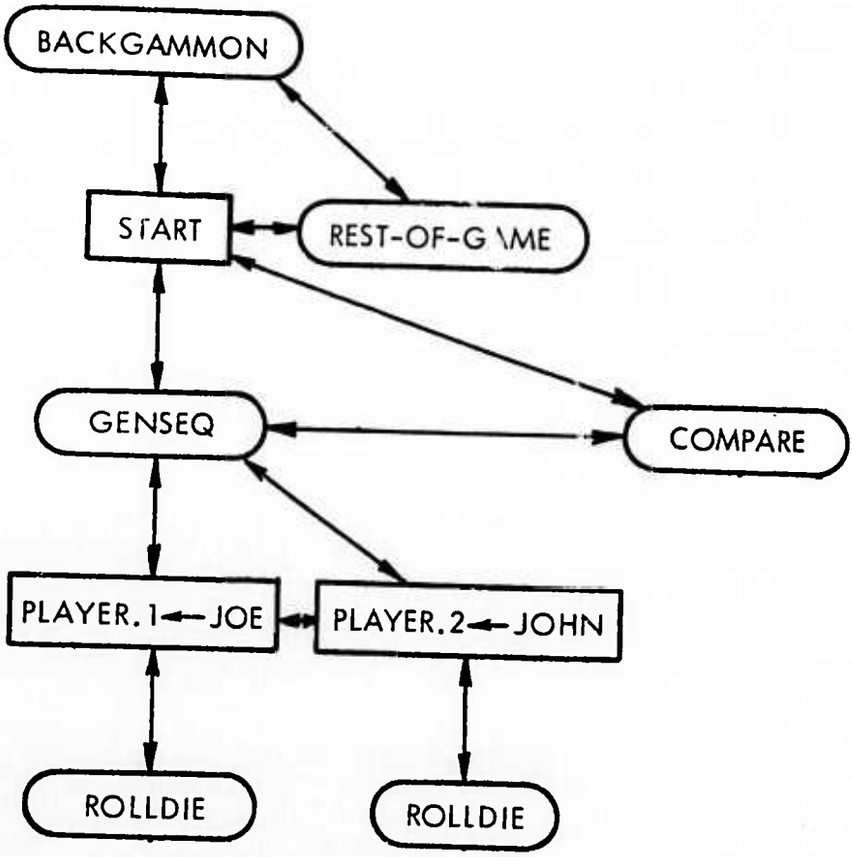


Figure 22. PEG after step 3

4. ROLLDIE, a nonterminal, produces the GENMEM event, which results in turn in a binding for DIEVAL.1. Since this GENMEM has no "next-event" in its definition, control passes normally to the ROLLDIE under PLAYER.2, where a binding for DIEVAL is chosen as before. Figure 23 shows the state of the PEG.

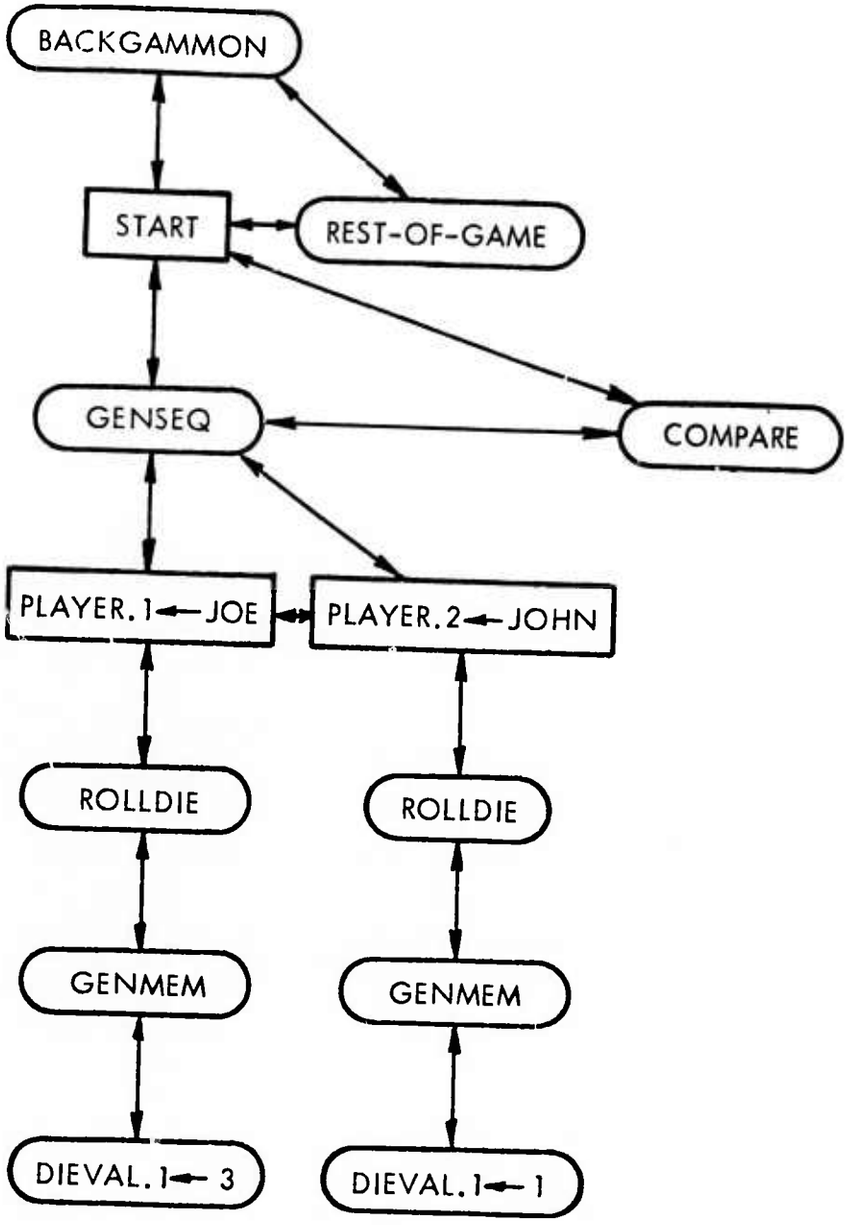


Figure 23. PEG after step 4

5. Control passes to COMPARE which causes the production

COMPARE := (INSERT PLAYER.(INDEX MAX DIEVAL)) -> FIRST-MOVE

to be appended to the PEG. When INSERT, another PLX primitive, is executed, PLAYER.1 is selected and reinserted into the PEG, shown in Figure 24.

6. When control passes to FIRST-MOVE,

FIRST-MOVE := (TERMINAL PLAYER.-1 'MOVES')

is chosen, passed to, and executed. Since INSERT is in FIRST-MOVE's access path, PLAYER.-1 (the last player mentioned) evaluates to JOE, producing the string "JOE MOVES" as the result of the TERMINAL event. Figure 25, now the same as the PEG given in Figure 5, shows the result of this action.

7. The program continues by moving to REST-OF-GAME, where some action presumably takes place, and concludes when the original BACKGAMMON event is crossed, leaving no unopened nodes.

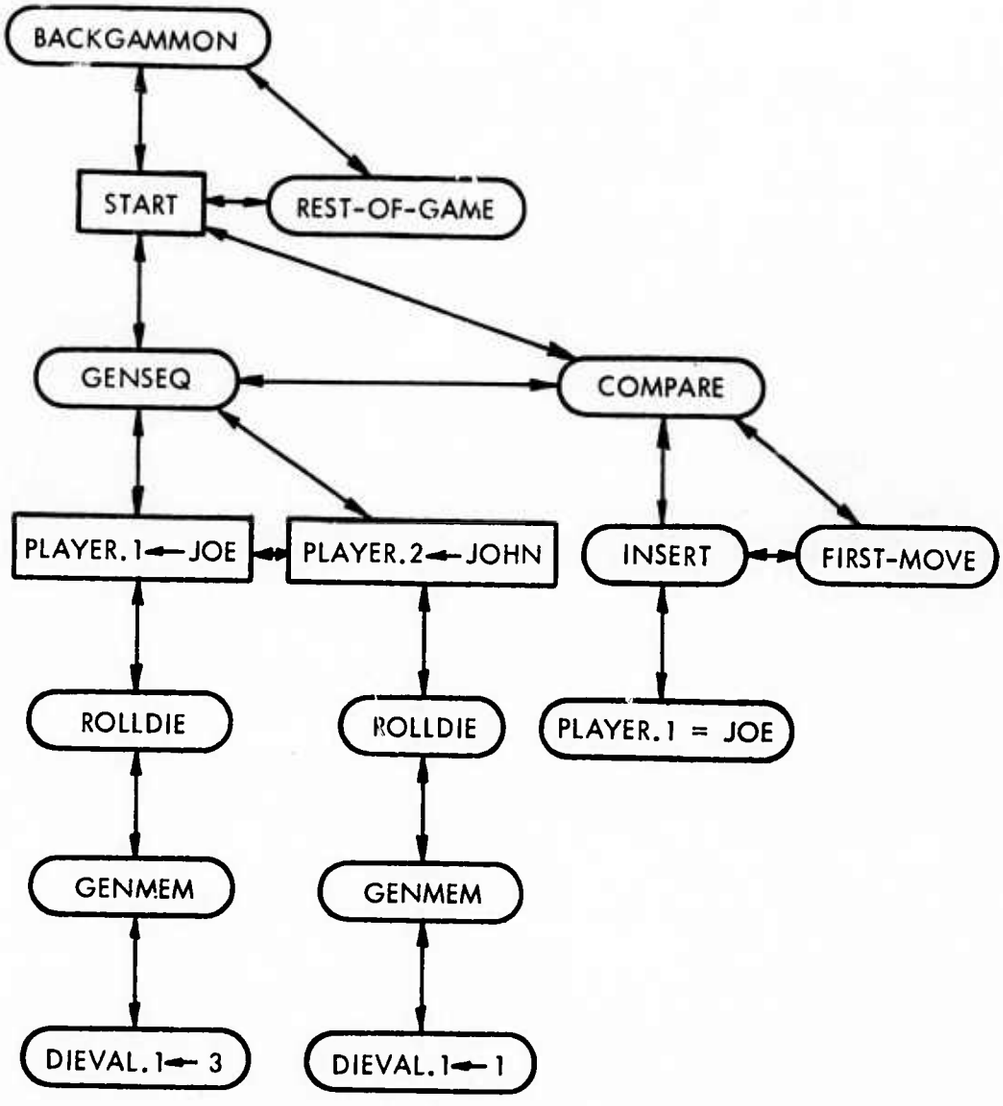


Figure 24. PEG after step 5

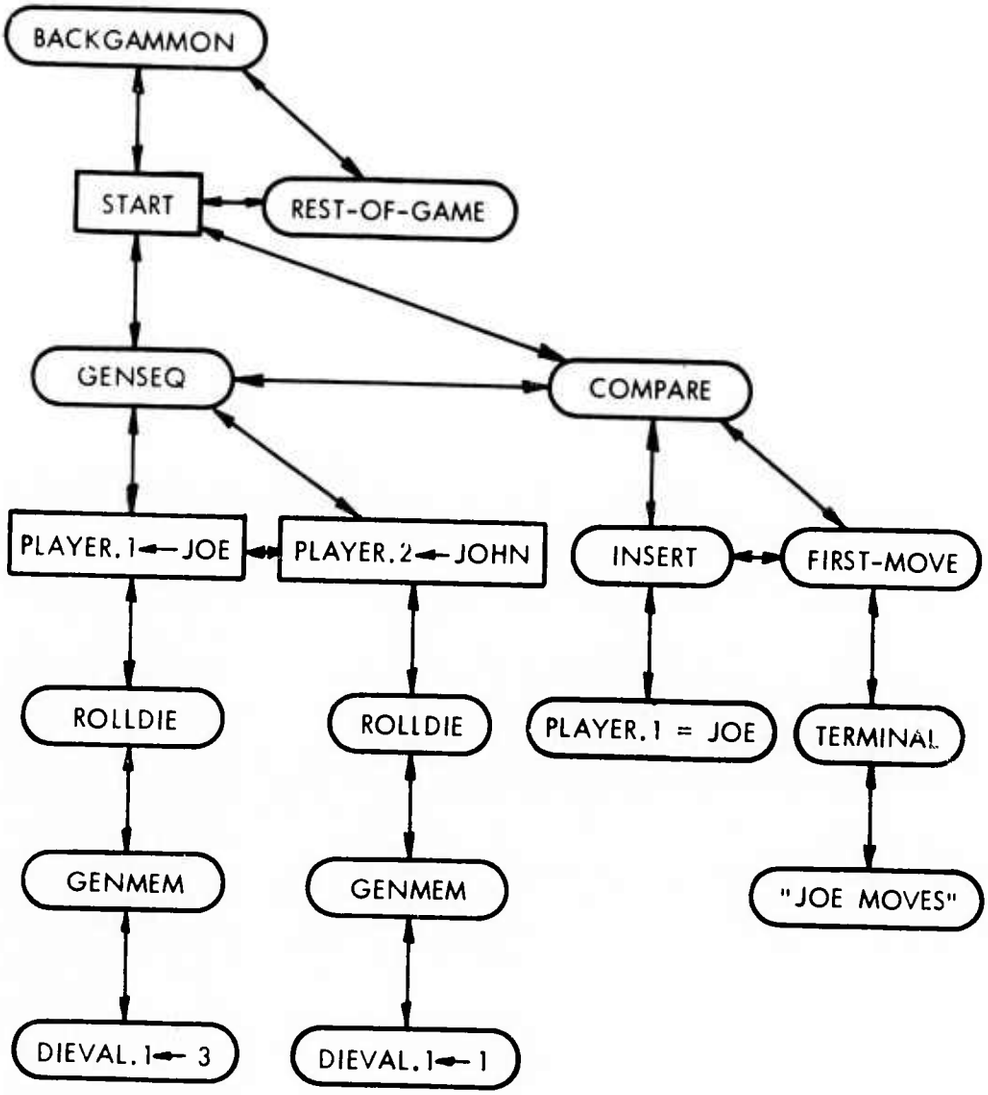


Figure 25. PEG after step 6

3.6 *SUMMARY*

Using graphs or trees as a medium for describing various properties of programming languages has been common in computer science research. For example, the Vienna Definition Language tries to formalize a method for stating a programming language's semantics by formulating an execution tree and providing primitives for manipulating that tree [WEGNER 72]. Each language construct is then defined in terms of these primitives and how they affect the execution tree, so that any implementation of the language will have a precise foundation. The tree is their mechanism for coordinating the entire formalism. Similarly, the PEG, by being the structure which defines a process, is the coordinating formalism within XREP.

In this chapter the production language was described by picturing each construct in terms of the PEG; the next chapter will study variables and access issues from the same viewpoint.

4. ACCESS PATH THEORY

4.1 INTRODUCTION

Access paths have been mentioned several times in preceding chapters, though they were underplayed in importance. The access path concept deals specifically with the evaluation environment presented to events, i.e., what data are available at any point during the execution of a program. The fundamental position taken by this study, that decisions should be made dynamically at execution time rather than statically at "compile" time, forces a thorough understanding of the execution model offered by the PEG. In this chapter the creation of that model will be analyzed from the standpoint of retrieving information from it, with the emphasis on the correspondence between English methods and those available through XREP. Studying the generation and retrieval of data permits the isolation of several problems which could occur in an Automatic Programming problem acquisition phase. This chapter will describe the generality of XREP's methods and the flexibility of the PEG, while the next will show how they can be used in debugging certain errors.

An event's access path was defined in Section 3.3 as the unique ancestor chain of events and nodes from it up to the root. Because it contains all the nonglobal data which may influence an event's behavior, the access path is the context of the current process, the structure which contains all relevant information. Proper program organization can be evaluated through an access path goal: an event should have access to no more or no less information than it needs to operate; and a deviation in either direction is a structural flaw in the program.

This study of access paths is important to Automatic Programming because human dialogue does not contain the explicit structuring found in formal programming languages, but implicit clues are found within its addressing mechanisms. By simulating these methods XREP tries to mirror the structure inherent in the original description. The definition and use of access paths in XREP enable the freedom of English usage to be reconciled with the rigidity of a programming language.

4.2 NATURAL LANGUAGE ACCESS METHODS

Natural language has three basic data retrieval methods: the unique name of the desired object is given, a pronomial reference is made, or the object is identified by type with some limiting

predicate. In the first case, a proper name is usually associated with the object: John, Chicago, USC, etc. In most cases this name is unique, any attached modifier is considered nonrestrictive or redundant and set off by commas as nonessential information. If necessary to the identification, the clause (now called restrictive and considered part of the noun phrase) is not enclosed in commas. For example, in the statement, "USC, which won the Rose Bowl in 1975, is private," the clause "which won the Rose Bowl in 1975" is nonrestrictive. However, if the statement becomes "the school which won the Rose Bowl in 1975 is private," the same clause is necessary to identify the subject, hence restrictive. Since most inanimate objects do not have proper names, the restrictive clause is a major retrieval mechanism and is the one with which XREP is concerned. A pronomial reference will later be shown to be a special case of the restrictive clause situation.

The various reference forms used in English are not easily classified, due to their scope and flexibility. Consider the following references:

1. The last player.
2. The first player.
3. The player who rolled a 5.
4. John's die value.
5. The player who rolled a die.
6. The die value rolled by the last player.
7. The last player before John.
8. The player who rolled the largest value.

The first two examples are common references made in situations where a particular type, in this case player, has more than one instantiation. Presumably the predicate, first or last, is needed because a reference to "the player" would be ambiguous. The numeric character of this predicate seems to be useful only in identifying end points of a type's members. References like "the second," "the next to last," or "the third" are not unusual, but "the sixth" or "fourth from last" are, since the possibility for error is greater for both the giver and receiver of the information. If someone had presented an unnumbered list to me and asked for a comment on the sixth item, I suspect that I would ask for confirmation of the item before commenting. Natural language does not often use complex computations like counting to six for specifying an object.

An alternative to counting is to "home" in on the desired object by giving extra information associated with the object. Examples 3 and 4 are of this type. In the former a player, the object of the search, is restricted by having 5 for his die value. If several players had the same number, more information would be necessary.

Example 4 is slightly different in that the die value is the object of interest. In both cases, notice that no explicit linkage is given to help make the proper association. In other words, John's

age could be requested in a context-free manner because all humans have an age, but not all humans have die values. The validity of "the last player's die value" depends on the environment of the request and, if proper, its resolution will be based on some proximity measure rather than some explicit rule. XREP's PEG allows exactly these kinds of associations to be made.

The player in example 5 is identified by association with an event as opposed to an object -- a situation hardly different from any of the preceding. However, notice that this form has no counterpart in traditional programming languages.

Examples 6 and 7 typify the relative types of addresses which replace the numeric kind. In the former "the last die value" might have sufficed, but its form emphasizes the player involved. In the latter counting was presumably not feasible, so a new context, John, is named and objects are referenced from this new focal point. This method is one of a class of naming mechanisms which is more elaborate and more context-dependent than those found in computer languages.

The last example is the most difficult because of the generality of the reference. It says to select a player based on the result of some function applied to an object associated with players. Many assumptions must be satisfied before such a request can be fulfilled: what is done if a player has no die value, what if a player has two die values, what if the result is not unique? Again, this request is highly dependent on the context of the inquiry; each anomalous case must be treated separately.

The examples given cannot possibly be exhaustive, but are intended to represent typical situations which arise in natural language. Each case will have an interpretation in XREP within the facilities of the production language.

4.3 *ACCESSING TYPED VARIABLES IN XREP*

Generation Numbers

A typed variable is created in XREP through a GENMEM or GENSEQ event. The value of the variable is assigned to the form

type.n+1

where n is the current generation number for this type in the event's access path. The generation number, defined in Section 3.3 as an integer which identifies the relative position of a variable, serves more as a convenience for the discussions than as a fundamental tool of the formalism because high generation numbers are not often used. As mentioned in the previous section, accesses to a set of types probably use numeric expressions only at the end points -- for example,

PLAYER.1, PLAYER.2, PLAYER.-1, PLAYER.-2 -- while accesses to the middle of such a group most probably name an intermediate target and then give relative specifications.

The scheme for assigning generation numbers is simple: for GENMEM the current number is incremented for a type; for a GENSEQ the numbers are incremented across the driving type. The assignment in a GENSEQ comes more from intuition and convenience than from a strong logical basis, since each of the elements could be assigned the same number. Figure 26 shows the GENSEQ from the BACKGAMMON game. On the left is the actual structure: the PLAYERS are numbered 1 and 2 (according to the GENSEQ rule) while each DIEVAL for the GENMEM is assigned 1, since each is the only DIEVAL in its own access path. The structure on the right of Figure 26 is also possible, since each PLAYER is likewise the only one in each corresponding access path.

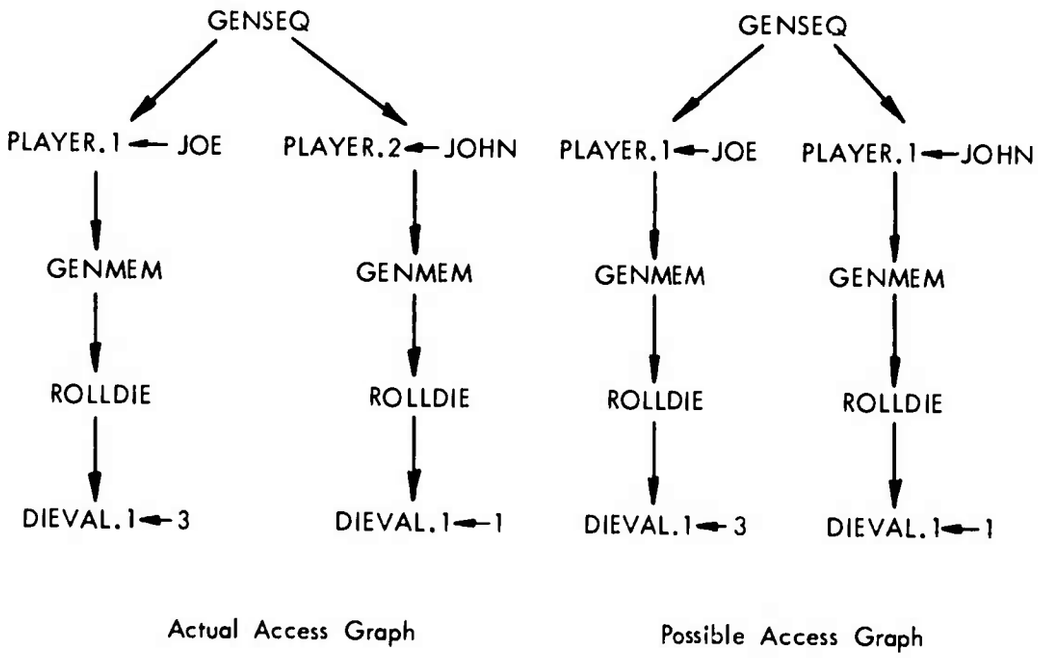


Figure 26. Generation number example

A more ambiguous situation occurs in the Access Graph skeleton shown in Figure 27. What should the last DIEVAL be numbered? A case could be made for 2, 3, or 4. A more complex numbering scheme involving extra indexing is also possible, but since this situation is rare and since XREP has many ways to access all the typed variables unambiguously without relying on the particular numbering schema chosen, this problem is one more of implementation than of substance. As a result this and similar anomalous situations will be downplayed; the emphasis will be placed on the addressing methods.

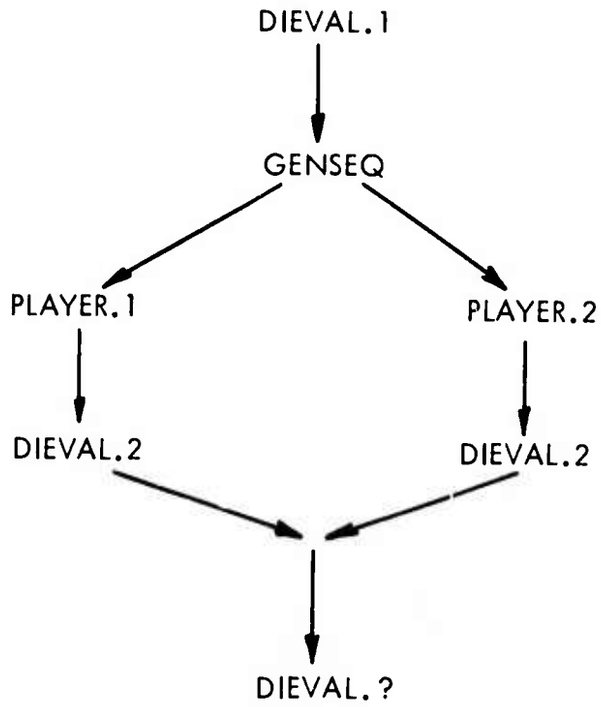


Figure 27. An anomalous generation number situation

4.4 *RELATIVE ADDRESSING*

One of the claims made earlier in this report was that the language and the PEG promoted a notion of spatiality for data items. That is, rather than merely a value, a variable also has a referenceable location within the evaluation environment. To take advantage of this extension, ways exist within the language of access data in a spatial manner.

The basic method is to refer to the variable type, together with an identifying expression as follows:

type.expression

The expression may be anything that evaluates to an integer (other than zero), or it may be a functional form, INDEX or FIND.

INDEX will be described in the next subsection as a function which inspects GENSEQ structures. FIND is a function which specifies a search for a type whose position is unknown. Its form is

type.(FIND AP1-expression)

For example, if a DIEVAL less than 5 is desired, the request is

.DIEVAL.(FIND (LT DIEVAL 5))

Other examples will be given later.

For the case in which "expression" of "type.expression" evaluates to an integer, the addressing interpretation depends on its value. If it is positive, that precise typed variable is looked for in the appropriate context path. This is a standard access, no different from traditional systems. If it is negative, then a relative access is defined from the point of this reference. For example, if PLAYER.-1 is the request, the value returned is the first PLAYER found in the search up the context path, i.e., the last PLAYER generated or inserted into the PEG. Similarly, PLAYER.-2 would refer to the second PLAYER in the search up the tree (the next-to-last player generated or inserted).

References of the latter type give the system its heterarchical flavor; different processes communicate in a nonhierarchical manner. Information is produced by a process and exposed to whoever has rights to it. A hierarchy is imposed only implicitly by the structure of the PEG in dealing with the scope of typed variables. This situation will allow us to reorganize programs with certain faulty retrieval attempts.

The negative generation number specifies an access relative to a reference point. Another kind is possible where the desired data is referenced relative to other data. Its form is

type.exp FROM spec {FROM spec}*

In other words, a valid reference is a type.exp followed by any number of "spec" separated by FROM, where "spec" is either an event name or another type.exp. The list associates to the right. Thus

DIEVAL.1 FROM PLAYER.-2 FROM ROLLDIE

is equivalent conceptually to

(DIEVAL.1 FROM (PLAYER.-2 FROM ROLLDIE))

though no parentheses are allowed, since any other structuring will not make sense. If a nonunique event is named in the access, the one "nearest" the current reference point is used.

When a typed variable which precedes a FROM has a positive generation number, it is located by searching down the same access path of the current reference point. In the above example, once PLAYER.-2 FROM ROLLDIE has been located, DIEVAL.1 specifies a downward search for the first DIEVAL encountered, not something named DIEVAL.1. Notice that if a GENSEQ structure (or any compound event) is passed in the "upward" search for PLAYER.-2 FROM ROLLDIE, the following downward search for DIEVAL.1 may be ambiguous, since each branch of the GENSEQ may contain a DIEVAL. The ambiguity of the situation, explicit and graphic, is easy to relate back to the user as an error.

To further emphasize how the FROM reference works, some hypothetical requests will be evaluated in the context of the Access Graph skeleton shown in Figure 28. Each reference will be given followed by an interpretation of its evaluation. Three items should first be reiterated: negative generation numbers are references up the Access Graph, positive generation numbers are references down the Access Graph, and no access strays out of the context path of the original reference point. The examples follow.

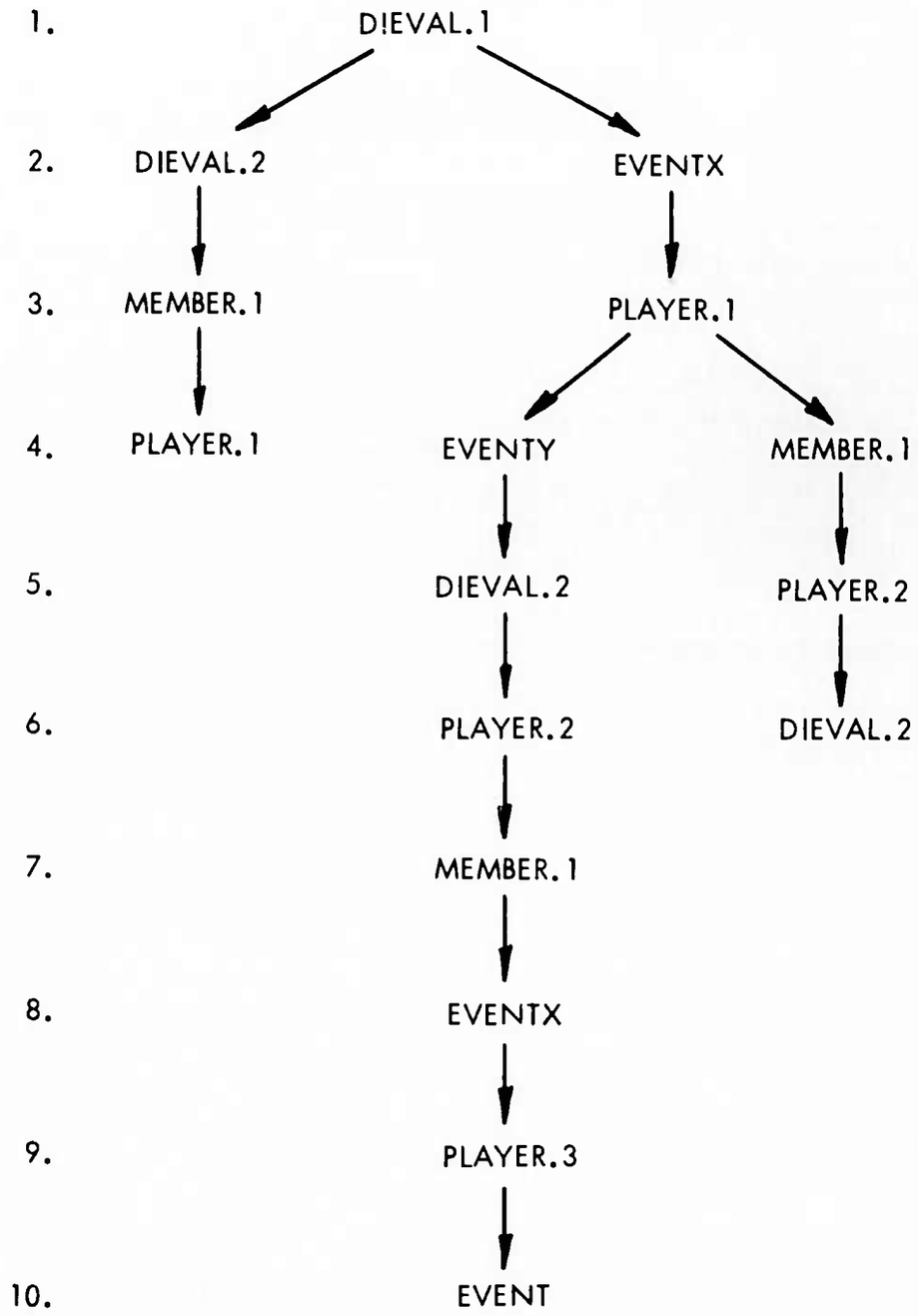


Figure 28. An Access Graph skeleton

Reference point: EVENT
Access request: PLAYER.-2 FROM MEMBER.-1

This reference is solved by locating MEMBER.-1, then finding PLAYER.-2 relative to it. MEMBER.-1 is found by looking up from EVENT for the nearest MEMBER, which happens to be MEMBER.1 of line 7. Using it as the new reference point, the new target, PLAYER.-2, evaluates to the PLAYER.1 of line 3. Note that if the request had been for PLAYER.-2 from EVENT, the result would have been PLAYER.2 in line 6.

Reference point: EVENT
Access request: DIEVAL.1 FROM PLAYER.-2 FROM MEMBER.-1

This request is initially the same as the one above, with PLAYER.-2 from MEMBER.-1 pointing us to PLAYER.1 on line 3. DIEVAL.1 from it means to now search down the access path for the first DIEVAL found, in this instance to DIEVAL.2 of line 5. Notice that the context path of the original reference point is not left, hence there is no ambiguity about downward searches.

Earlier it was mentioned that this string of FROM references associates from the right. It is easy to see why, if you try to evaluate the above request from left to right.

Reference point: EVENT
Access request: PLAYER.1 FROM EVENTX

In this request EVENTX in line 8 is located (not the one in line 2), with PLAYER.1 from it resulting in the PLAYER.3 of line 9.

Within this framework the example English references given in Section 4.2 can now be translated.

1. The last player.
   PLAYER.-1

2. The first player.
   PLAYER.1

3. The player who rolled a 5.
   PLAYER.-1 FROM DIEVAL.(FIND (EQ DIEVAL 5))

4. John's die value.
   DIEVAL.1 FROM PLAYER.(FIND (EQ PLAYER JOHN))

5. The player who rolled a die.
   PLAYER.-1 FROM ROLLDIE

6. The die value rolled by the last player.
   DIEVAL.1 FROM PLAYER.-1

7. The last player before John.
   PLAYER.-1 FROM PLAYER.(FIND (EQ PLAYER JOHN))

The reference to "the player with the largest die value" will be examined in the next subsection.

Addressing a GENSEQ

Thus far all the access questions have ignored the GENSEQ node. Since it represents a structure of independent events, some mechanism must acknowledge the coherent character of the node. Basically, the GENSEQ can be thought of as being a set of contexts or symbol tables which contain data. Thus a request from outside the GENSEQ (but in the same access path) may wish to get a "pointer" to a branch (context) of the node in order to do some calculation. The INDEX function accomplishes this task.

The call to INDEX is

maintype.(INDEX function subtype subtype-depth)

where "maintype" is the generator type of some GENSEQ, "function" is used to select a member of the set "subtype," "subtype" is some type which appears in each branch of the GENSEQ in question, and "subtype-depth" gives the relative position of "subtype" to "maintype." The "subtype-depth" defaults to one if it is not specified. It gives, as mentioned above, a relative position. For example, if it is 2, the located subtype satisfies

subtype.2 FROM maintype

for each GENSEQ branch.

In the BACKGAMMON program, the COMPARE rule was

COMPARE := (INSERT PLAYER.(INDEX MAX DIEVAL)) -> FIRST-MOVE

The "maintype" is PLAYER, the "function" is MAX, the "subtype" is DIEVAL, and "subtype-depth" is 1 since it was not specified. Notice that if each player rolled two die and the comparison was to take place on the second roll, the call would be

PLAYER.(INDEX MAX DIEVAL 2)

In the COMPARE rule the segment PLAYER.(INDEX MAX DIEVAL) tries to get a pointer into the GENSEQ node to the player with the largest die value. The result of this access must be unique; otherwise a failure which leads to backup occurs. Most often INDEX will be used in conjunction with INSERT in order to provide a context-maintaining pointer for future references. The reinsertion of the found type in the PEG is implemented as an indirect pointer back to the original binding. So once INSERT has done its work, a reference like

DIEVAL.1 FROM PLAYER.-1

will result in the largest DIEVAL (just found by the INDEX function).

The goal of the INDEX function is straightforward, but the complexity of its parameters is not. The decision to make it work on a "subtype" via one "function" is arbitrary but not restrictive due to the arbitrary power which can be programmed into "function." In any case the situation is not critical, since perspicuity (not to be underrated) and not capability is at stake.

A different design problem can be captured by viewing Figure 29. What should the result of a DIEVAL.-2 reference from EVENT be? This situation is so obscure that the time spent on it may not be worthwhile; a case could be made for any of the first three. Most likely this particular graph will never exist, and if it does a more specific access would probably be made. In Section 4.6 a precise formulation of access paths will be given; this question will be answered then.

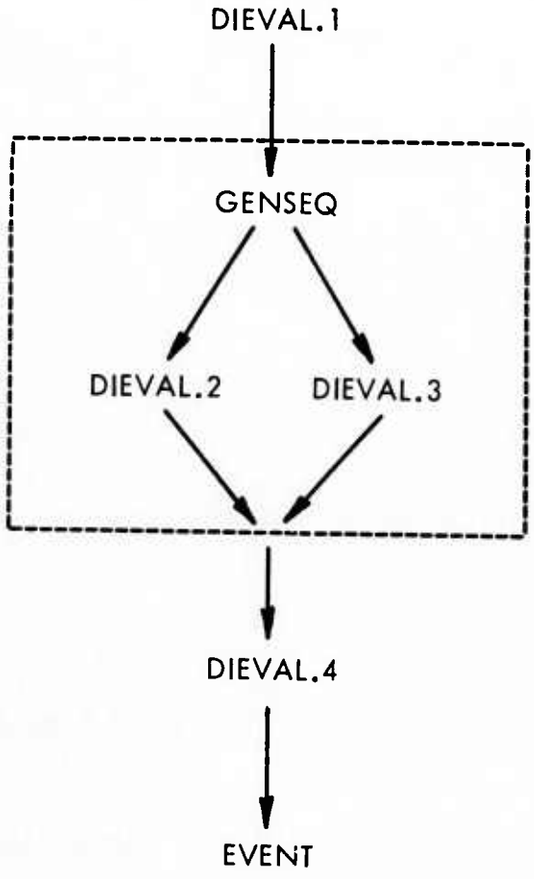


Figure 29. An Access Graph skeleton

4.5 *THE ACCESS PATH PROBLEM*

A study of algorithms meant for humans (rules for games, directions for product use, etc.) reveals that information tends to come in functional packets without regard for any structuring issues. In trying to code such specifications for a computer, programmers often produce a product which reflecting diffuse structure, global variables, uncontrolled transfers, all items which Dijkstra

deals with in his structured programming theory [DIJKSTRA 72]. His ideas present a unifying goal to programming, but are not at all natural for humans, programmers included. Yet human specifications do have structure, though much of it is implicit. The use of anaphoric and relative references, ellipses, and sequential information provide clues that human dialogue and descriptions contain structure, though perhaps not as formally as one of Dijkstra's structured programs.

Consider the Automatic Programming task. Assuming that one of its goals is to find and maintain the structure inherent in the natural language input, the access path problem is to organize the fragmented problem statement so that during execution every process has access only to relevant information while maintaining the appropriate sequence of actions. Given that goal, some general issues can be discussed.

Heterarchy versus Hierarchy

Automatic Programming has found its way into the realm of Artificial Intelligence because of its general problem-solving character. This brings with it all the techniques and design issues normally found in Artificial Intelligence, heuristics, search, representation, etc. Program organization, as a representation problem, is one of these concerns.

A strict, pure hierarchy which defines a structured program may not be realistic or even desirable as a target for preliminary programs generated by Automatic Programming due to the nonhierarchical nature of the human input. The heterarchical ideas, mentioned in Section 2.3, offers the flexibility necessary in the initial translation attempt. In this framework control is diffuse, processes are activated in a goal-oriented manner based on the state of the computation, while its data "exists" and is found as needed.

CASAP [BALZER 73] tests these system ideas in a simple card playing environment. Its basic feature comes from the interface between a routine and the data base. In CASAP a process requests some information, with the interface trying to find it, thus centralizing the knowledge about the data base.

Whether that much flexibility is needed is open to question. XREP takes a middle position between the two extremes of heterarchy and hierarchy by offering a nonprocedural control flow, yet addressing data access and scope issues.

Nonprocedural Control Flow in XREP

The control flow in XREP is determined by the production system character of PLX, organized but not procedurally oriented. The segmented nature of production rules are like procedure or subroutine calls but without a formal parameter-passing mechanism. This model has two purposes: to provide a control which can be easily monitored and understood and at the same time to allow the

freedom and flexibility gained by giving up formal parameters. These qualities do not, however, come at the expense of requiring all data to be global.

Access and Scope of Data

By generating data and inserting it into the PEG, GENMEM, GENSEQ, and INSERT create an evaluation environment typical of ALGOL-like languages. Most production systems do not have this capability. Instead, they operate out of a strictly global data base, a mode which results in well-documented flaws for programming systems(i). The access paths of the PEG give its data scope and a specific place within a process, features which will make PLX programs amenable to analysis.

Determining Access Paths

Since the PEG represents a complete history of the process, dynamic bindings included, XREP has some basis for determining the proper structuring of the data. If the sequencing of the program is correct, organizing the access paths is a realistic and desirable goal. The information is there; the system needs only to find and organize it. Chapter 5 will discuss the problem in more detail.

The access path is the fundamental concept behind most of the debugging efforts of XREP. Thus far only the Access Graph has been given for viewing access paths. The next section will give the formal methods of determining them.

4.6 *COMPUTING ACCESS PATHS*

In the definition of an access path, both events and nodes were mentioned in order to emphasize their difference. A node, defined as the execution of a compound event, has a simple interpretation when calculating access paths in the PEG and only a slightly more complicated one when viewing the Access Graph.

The Access Graph in Figure 30 is the same as that of Figure 4 except that the compound events are boxed in dotted lines. To find an access path for an event, move up the tree, include whatever is found without "entering" nodes (i.e., they are to be treated as entities). Thus the access path for INSERT is

COMPARE, GENSEQ-NODE, START, BACKGAMMON

(i) See [WULF 73] for a discussion of this topic.

while the path for the second GENMEM is

ROLLDIE, PLAYER.2<-JOHN, GENSEQ, START, BACKGAMMON

Treated in this node format there is no ambiguity in calculating access paths because every event has a unique ancestor; the "joins" in the Access Graph occur only within nodes, which are masked when viewed from outside the node.

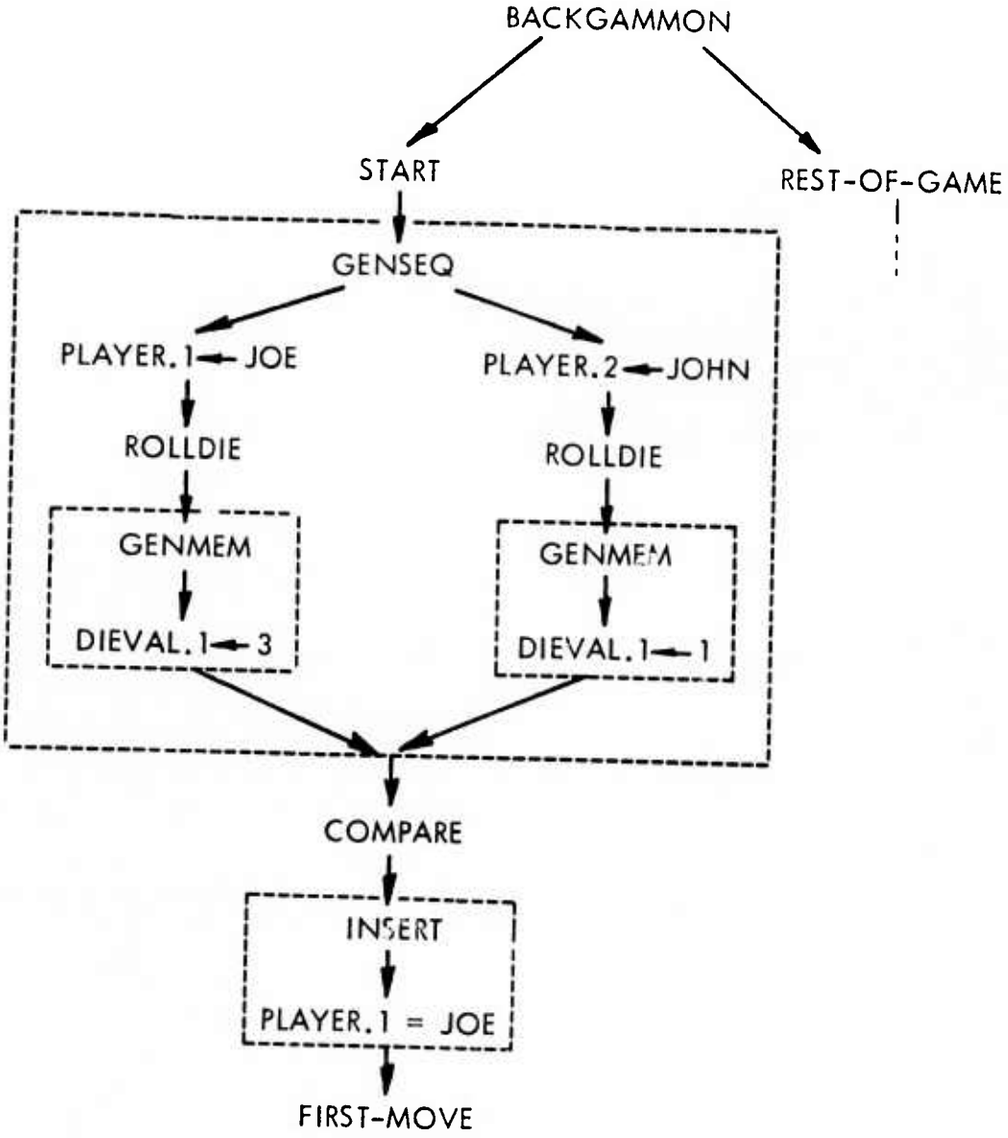


Figure 30. An Access Graph in node format

This situation is simpler when considering the PEG. The access path of an event is determined by moving left or up (when no left link exists), with all events included except those which are protected (rectangular) and reached by a left link. In Figure 5 (repeated for convenience) the access path for INSERT is

COMPARE, GENSEQ, START, BACKGAMMON

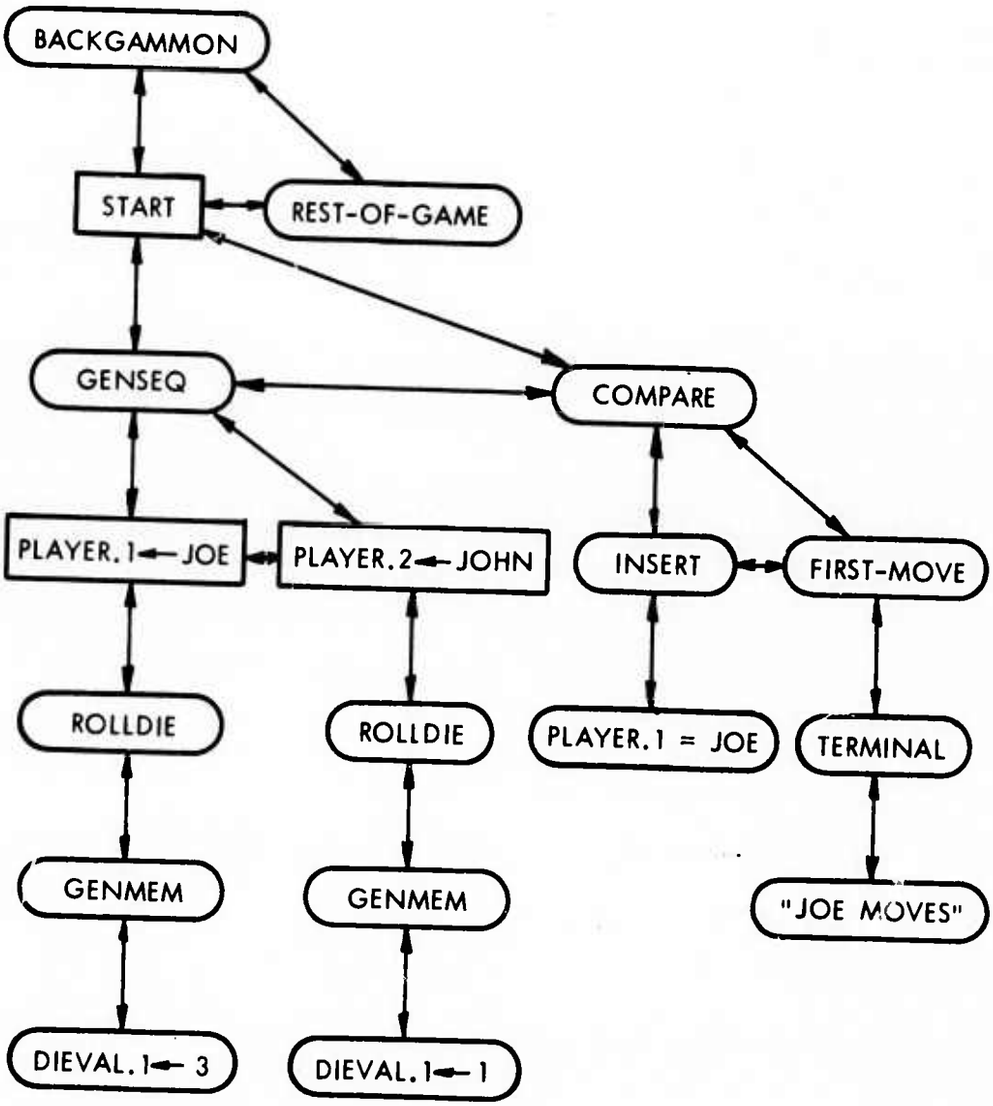


Figure 5. A Process Elaboration Graph (PEG)

START, though protected, is included because it was reached by an up link, not a left one. Similarly, the second GENMEM visits

ROLLDIE, PLAYER.2<-JOHN, PLAYER.1<-JOE,
GENSEQ, START, BACKGAMMON

but PLAYER.1<-JOE is removed, since it is protected and was reached by a left link, leaving

ROLLDIE, PLAYER.2<-JOHN, GENSEQ, START, BACKGAMMON

as its access path as before. The boxing operation done to the Access Graph is reflected in the PEG by never moving "down" in tracing access paths since the down link represents the execution of the compound event.

Still, once in an event's access path, a node may be inspected under the proper conditions. For example, the INDEX function was designed for just that purpose. However, the question raised in Section 4.3 concerning a PLAYER.-2 request to the graph in Figure 29 forces the definition of those proper conditions.

The decision, arbitrary in some sense, was to allow any descendant of a compound event access to its information. It only remained to decide how to "linearize" a compound event for inclusion in the access path. The logical choice is to consider the node in reverse time sequence. For the GENSEQ node in Figure 30, that sequence is

DIEVAL.1<-1, GENMEM, ROLLDIE, PLAYER.2<-JOHN,
DIEVAL.1<-3, GENMEM, ROLLDIE, PLAYER.1<-JOE

The algorithm for this process is

LINEARIZE (NODE)

1. If NODE is NIL then return.
2. LINEARIZE each son of NODE, rightmost son first.
3. Print NODE

When a node has no sons, then the recursive call in line 2 will be LINEARIZE (NIL); hence the test in line 1.

In an ALGOLized version of LISP, the algorithm is

```
(LINEARIZE
  [LAMBDA (NODE)
    (if NODE=NIL
        then NIL
      else if NODE:RIGHT-LINK EXISTS
        then (APPEND (LINEARIZE NODE:RIGHT-LINK)
                     (APPEND (LINEARIZE NODE:DOWN-LINK)
                             (LIST NODE)))
      else (APPEND (LINEARIZE NODE:DOWN-LINK (LIST NODE])
```

Without going into a great deal of LISP detail, the second "if" clause generates a list of events by moving across a level of the PEG. The order of the APPENDs produces this list in reverse order, first the NODE's right-link, then the NODE's down-link, and finally the node itself. The recursion, of course, takes care of embedded nodes, while the last "else" clause handles the last-son condition.

With LINEARIZE defined, a complete access path algorithm can be given.

```
(ACCESS-PATH
  [LAMBDA (NODE)
    (if NODE=ROOT
        then NIL
      else if NODE:LEFT-LINK EXISTS
        then (APPEND (EVALUATE NODE:LEFT-LINK)
                     (ACCESS-PATH NODE:LEFT-LINK))
      else (CONS NODE:UP-LINK (ACCESS-PATH NODE:UP-LINK])
```

The simple function EVALUATE is defined to be

```
(EVALUATE
  [LAMBDA (NODE)
    (if NODE is PROTECTED
        then NIL
      else (LINEARIZE NODE])
```

It merely eliminates protected nodes reached by a left-link.

Notice that the last "else" of ACCESS-PATH clause handled the up-link case. In this situation, the event is added (accomplished by LISP's CONS) to the list only, not operated on by LINEARIZE. Applying ACCESS-PATH to INSERT gives

COMPARE, LINEARIZE (GENSEQ), START, BACKGAMMON

as desired, while applying it to the second GENMEM produces the same result as before.

4.7 *THE PEG AND OTHER EXECUTION MODELS*

The semantics of a program executing within XREP are captured by the PEG in depicting all the control and access issues. From this standpoint XREP's execution is similar to that found in any language which operates out of a stack, like LISP or ALGOL. But the role intended for the PEG is more diversified.

In describing his Contour Model as a structure which defines execution of block structured programs, Johnston mentions that one of its features is that algorithms and records of execution are separate but related components [JOHNSTON 71]. His picture of execution as a contour was specifically designed to give precise meaning to all phases of execution of block structured programs, including passing control and accessibility of data. The model is separate from the program and can be interrogated independently.

The PEG has the same flavor. It is independent of the PLX program but yet is carefully designed to capture the structure of the production rules of which it is comprised. Like the Contour Model, a visual display of execution, available for analysis, can be related back to the original program. The intent, however, is not semantic definition, but understanding and debugging. This role accounts for the inefficiency in never deallocating any completed processes, a major concern of other execution models. The entire history of the execution is needed for debugging purposes.

In XREP access paths are implemented in a configurable way. In other words, each node in the PEG is semiautonomous, a result of the segmented nature of production rules. If access is needed to a different node, then a link, by way of the event separators, must be built between the two through the access path mechanism. The "independence" of nodes gives the debugger specific entities on which to address the access path problem. By building or inspecting the links between nodes, the program can be properly structured. In SIMULA [DAHL 66], an ALGOL-based simulation language, a similar situation exists in linking processes together. A process in SIMULA is meant to be a complete action acting on its local data and on data generated and stored within other processes. The linkage between processes is set up by items called "elements" declared within the requesting process. The thought behind this organization concerns the needs of simulation systems. The demands of simulation makes it difficult to program problems in standard languages. The individual, nonhierarchical nature of a SIMULA process, together with explicit programmable links, evidently better reflects simulation situations.

The event separators in PLX are very similar to the "elements" of SIMULA; both have the same basis. In SIMULA processes are best described as separate entities, while in PLX the production rules are also meant to be independent in nature. Both systems needed a way to get the separate process to communicate; SIMULA uses a specific pointer, while XREP does it by configuring access paths.

All the features and capabilities claimed for PLX and the PEG are directed toward providing an internal model which captures algorithms acquired by an Automatic Programming system from a natural language source, and which is amenable to debugging analysis. The access path issues of this chapter, though emphasizing the former concerns, prepared the groundwork for many of the debugging algorithms of the next chapter.

5. INTENTIONS AND DEBUGGING

5.1 INTRODUCTION

Thus far, XREP has been described as a system composed of various programming constructs which combine to provide an environment suitable to Automatic Programming. The overall goal is of course to generate (or write) correct programs. In this direction this dissertation starts from an existing program, uses expectations in the form of intention strings, then automatically debugs certain errors that arise during execution. Before the details of XREP's intention and debugging mechanisms are described, other works that address the problem of writing correct and reliable programs will be reviewed: Section 5.2 describes "standard" programming systems, Section 5.3 describes aspects of the program proving process, while Section 5.4 covers automatic debugging systems. Each section will emphasize the expectations of a system, how they are received, and how the system uses them to help find and correct errors.

5.2 SYSTEMS FOR WRITING PROGRAMS

Systems which provide environments for programmers deal with a class of errors normally associated with programming details -- bad syntax, misspellings, etc. They fall into two basic groups: static compile-time errors and dynamic run-time ones. In trying to cope with either set, the software system can only assume or expect that the programmer's input is rational, and that any simple, detectable, obvious error should be repaired if possible.

In a batch environment the only expectation held by a compiler is that the input is intended to make sense. So when PL/I repairs a program by inserting missing semicolons, progress has been made, beyond the infuriating FORTRAN error message that

```
GO TO (10,20,30,40) I
```

is missing a comma after the right parenthesis. In any case, purely syntactic errors are not that interesting in this environment, while run-time ones only cause immediate failure.

INTERLISP [TEITELMAN 74] will be the model for the discussion of what can be done by an

intelligent on-line system. In addition to providing a series of interactive debugging tools(i), INTERLISP provides a LISP-oriented editor and an automatic error correction package named DWIM (Do What I Mean) [TEITELMAN 73]. DWIM does automatic spelling correction, syntax modification, and the like based on run-time analysis of program errors. DWIM works well because it knows about LISP.

In his Ph.D. dissertation, Yonke develops a system which has similar capabilities, but which is language-independent, since it is driven by external language specifications (written by an expert) [YONKE 75]. Both systems show what can be accomplished if the computer is allowed to be more active in the program construction process, even though working in a task-independent domain.

5.3 *PROGRAM PROVING SYSTEMS*

The intent of the program-proving community is to provide formal methods for verifying programs. In their framework a program is augmented with strategically placed assertions for describing what should be happening at various points in the process. From these assertions, verification conditions can automatically be generated and then proved in various ways(ii).

The assertion language in current research is typically some dialect of the first-order predicate calculus. Its role is to provide a second description of the program, where the program itself is the first -- the assertions are thus redundant specifications of the program. On the assumption that its assertions are an accurate statement of the programmer's intention, the program is formally proved by verifying that every execution will satisfy all the assertive conditions. The assertions are thus the system's expectations for the program(iii).

One problem with this technique is that the first-order predicate calculus is not very expressive and can produce detail in almost incomprehensible quantities. Trying to prove the verification conditions becomes a large task, difficult for either man or machine. An interactive theorem prover is used in one program verification system so that the user can help guide the proofs [GOOD 75b].

What seems to be needed is higher levels of abstraction in the assertion language, or whatever level of description satisfies the user. If a program is to be proved, a description of its

---

(i) See [MANN 73] for a survey of these debugging tools.

(ii) See [ELSPAS 72] for a complete review of this process.

(iii) In fact, Buchanan and Luckham automatically generate some simple programs from assertions and an appropriate set of axioms [BUCHANAN 74]. The formal method of generation, based on theorem-proving techniques, guarantees that the resulting program is correct.

intent is mandatory. However, if it is going to be as difficult to write assertions as it is to write programs, then the cost and feasibility of this process are open to question.

D. Good recognizes this problem and suggests a programming environment in which programs and assertions are stepwise refined together from the start of the development phase [GOOD 75a]. Thus a program can be proved at various levels of abstraction. For this idea to work, expressive, formal assertion languages are needed.

> If we are to construct proved programs of significant size and complexity, then it seems that we should . . . state precise specifications. Obviously, we must be able to state the specifications before we can prove that the program meets them. Although some progress has been made in this area. . .stating specifications for a program remains a difficult problem.(iv)

Many of the phases of program proving are being automated in one form or another. One that has just begun is automatic debugging. Many difficult problems need to be solved before automatic debugging is realistic. The next section will show the complexity of some systems which do attempt it in some restricted domain.

5.4 *AUTOMATIC DEBUGGING PROGRAMS*

The three programs reviewed in this section come from MIT, each with the flavor of Artificial Intelligence. Each attempts to solve complex tasks within a well specified domain by applying its "expertise" to problem situations. The automatic correction accomplished by each reflects a deep understanding of the associated problems.

In his dissertation about understanding LOGO(v) programs, Goldstein uses an external model language to describe the intent of a picture [GOLDSTEIN 74]. The picture drawn by the accompanying LOGO program is then matched against the model. If a difference is detected between the diagram and the model, debugging occurs.

Figure 31 shows how to describe a simple line drawing of a tree in the model language. Figure 32 shows a LOGO program whose intent is to draw that tree together with its result. Several model violations are readily apparent, specifically M4, M5, and M7. Using those violations as its debugging impetus, his system produces the converted program in Figure 33.

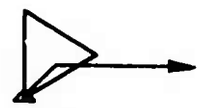
(iv) D. Good, "Provable Programming." ***International Conference on Reliable Software***, Los Angeles, April 1975, pg. 411.

(v) LOGO is a graphics system, devised by Seymour Papert, intended for children.

```
MODEL TREE
M1 PARTS TOP TRUNK
M2 LINE TRUNK
M3 EQUITRI TOP
M4 VERTICAL TRUNK
M5 COMPLETELY-BELOW TRUNK TOP
M6 CONNECTED TOP TRUNK
M7 HORIZONTAL (BOTTOM (SIDE UP))
END
```

Figure 31. A Goldstein tree model

```
TO TREE1                ; version 1
10 TRIANGLE             <- (accomplish top)
20 RIGHT 50             <- (setup heading)
30 FORWARD 50           <- (setup position)
40 RIGHT 50             <- (setup heading)
50 FORWARD 100          <- (accomplish trunk)
END
```



TREE 1
VERSION 1

Figure 32. Incorrect tree program

```
TO TREE1                ; version 4
 5 RIGHT 30             <- (setup heading such-that (horizontal (side 3 top)))
                        <- (assume (enter TREE1 statement 5) (= :heading 0))
10 TRIANGLE             <- (accomplish top)
20 RIGHT 60             <- (setup heading)
30 FORWARD 50           <- (retrace (side 3 top))
40 RIGHT 90             <- (setup heading such-that (vertical trunk))
50 FORWARD 100          <- (accomplish trunk)
END
```

Figure 33. Corrected tree program

Notice that the model language is different from the assertion concept of the last section in that it describes the output of the program, not intermediate states. The adequacy of the model language is, however, hard to ascertain, since the class of programs handled by the system does not allow conditionals, variables, recursion, or iteration. The complexity of the debugging effort is somewhat foreboding, considering these restrictions.

In his work on model debugging, Bill Mark has a less formal intention language [MARK 74]. In his system the user specifies a "goal" along with the model. If the goal is not attained during simulation of the model, it is debugged with that goal as its driving target. For example, a particular business model may be set up which is expected to make six sales. If only five are made, the user gives the system

(GOAL (INCREASE SALE 1))

as it goes into a debugging mode to find the cause of the failure.

Gerald Sussman's dissertation has the character of both Goldstein's and Mark's programs. Goldstein's effort is classical in intent: a flawed *program* is debugged. In Mark's work a model of assertions is modified when unintended interactions prevent the goal from being attained. Sussman's HACKER [SUSSMAN 73] is given a goal, but in the problem solving framework of the blocks world. In solving the problem HACKER tries to find an applicable program; if none is available it writes one. In either case, execution of the program may manifest a bug which HACKER will try to resolve if possible. HACKER's intent is not to just solve any specific problem, but to write generalizable programs for handling a class of related problems.

For example, in Figure 34 the scene with two blocks and a table is the setting for the request.

(MAKE (ON A B))

HACKER finds a simple program to do it.

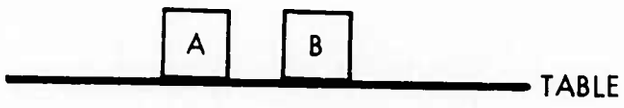


Figure 34. Scene for (MAKE (ON A B))

If the same request is made for the scene in Figure 35, a bug occurs, since the simple program cannot move A since C is on it. The program manipulator receives the error message and patches the performance program so that C is first put on the table and then A is put on B.

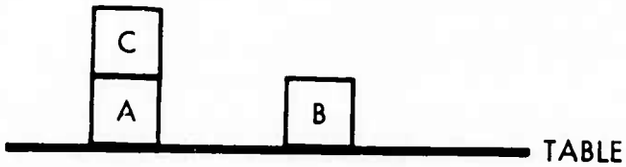


Figure 35. A more complex scene for (MAKE (ON A B))

The modification to the resulting program is general enough to solve the same problem for the scene in Figure 36.

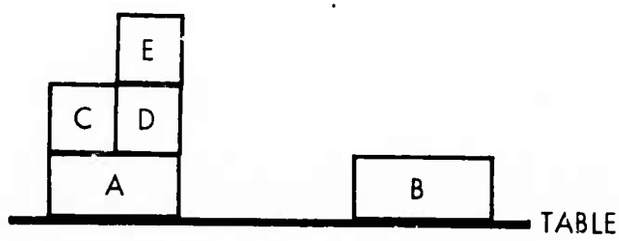


Figure 36. A generalized scene

HACKER basically handles three types of errors: unsatisfied prerequisites (eg., the case above), protection violations (eg., a subgoal is undone), and violation of domain-specific "aesthetic" principles (eg., moving the same block twice) by storing information about them in many different system modules. The attack on a bug is a complex dialogue between various independent system components, each with expertise in a specific area. The closed, noninteractive nature of HACKER is impressive in performance but perhaps causes unnecessary complexities in the general program debugging task.

Each of these theses enters the new automatic debugging area in a familiar way, with specific problems in specific domains. However, each derives techniques which can be extracted from the work and incorporated in an interactive Automatic Programming type of system as a set of possibilities for trial and discussion. The immediate impact of these works will come not from results within closed systems where myriads of second-order problems obscure their possible contributions, but from their availability as high-level debugging tools within smart software environments.

5.5 *XREP'S INTENTION STRING MECHANISM*

As described in Section 2.5, program expectations are communicated to XREP via a string of discrete events which are to be matched against the results of the TERMINAL events. When a match fails, a potential bug may have been uncovered. The system tries to identify it, relate its findings to the user, then suggest a particular debugging technique. Used this way, the intention string provides an interface between XREP and the user in a form usable by both. The power of this method comes from the freedom and flexibility of both the content of the TERMINAL statement and its placement within the program. If either is restricted, then generality is lost in potential stages of program development.

In the program-proving work correct placement of the assertion statements is fundamental; all the analysis depends on proper sequencing through those statements. But only the first-order predicate calculus assertions are allowed thus far, forcing the user into detail in which he may not be interested. The quotation from Good on page 70 indicates that the range of possible program descriptions must be expanded if they are to contribute to all stages of a program's development. XREP has that capability but without the security of formal proof procedures.

The assertion language in Goldstein's LOGO work is restricted in both dimensions. Besides requiring full descriptions of the completed picture, his system makes it completely external to the target program. This separation may be clean, but it seems to add unnecessary complexity to his analysis, since he must derive all the sequencing information himself. The modularity in picture composing makes the program segments functionally related to part of the overall diagram, so that the corresponding assertions have a well-defined place in the program while general statements could be located at the beginning of the program, much like the program-proving work. The "<-" comments on each line of his LOGO code perform exactly these functions, adding yet a third source of redundant information, one which relates the declarative model to the actual program. Actually the amount of redundancy Goldstein requires should be an omen for Automatic Programming work; his method, with three descriptive information sources, may be exactly the right one for proper program generation and debugging.

Comparing XREP's intention string mechanism to Sussman's HACKER is difficult. The main source of errors handled by Sussman are those which can be classified as implicit program specification errors. In other words a program which HACKER uses or writes is expected to conform to specifications located in various system modules. While the system pursues its goal statement, those modules may complain that they are being violated. The debugger tests the validity of their claim, taking the appropriate action. The direction for all actions comes from the goal statements.

Sussman's goal statements, like (MAKE (ON A B)), are very much like the declarative models in Goldstein's work. They are simple in form but more difficult to attain because Sussman tries to generalize learned techniques, handles recursive and iterative situations, and writes the programs automatically. His simple primitives allow more complex processing, while Goldstein, in the richer LOGO language, must restrict himself to a LOGO subset to achieve comparable results. Still, Sussman works hard at doing all the processing automatically, generating all his own subgoals and so forth. He has no way of specifying intermediate results, which may aid in the problem solving.

The philosophy of XREP, which is not in the problem-solving business until debugging time, is that any information useful to producing a program should somehow be able to be incorporated into the program description, whether it be intermediate results, general information about the state of the process, or just redundant material. The intention mechanism will have many of those capabilities.

Remember that the intention string not only aids the debugging process, but also guides the top-down execution of PLX programs. The nondeterministic choice points in PLX, either rule choices or GENMEMs, are forced down the the right path in trying to match the intention string. If a GENMEM generates a die value like 4 and the intention string expects 5, the execution will back up to that GENMEM until a 4 is generated. Similarly, a bad rule choice will be undone in order to try to match the current state of the string. This is the nature of the parsing operation.

The remaining sections of this chapter will discuss various program errors ard the debugging effort necessary to solve them. In most cases, the intention string will be fundamental in the detection of bugs, while also providing the necessary information to the debugger to properly modify the program to alleviate the problem.

5.6 *GENERAL ERROR DISCUSSION*

The most difficult task in automatic debugging is determining exactly where an error has occurred. The manifestation of a bug usually is obvious; assigning responsibility is much more difficult (witness the extraordinarily complex control structures of the debugging programs). If backtracking, automatic or otherwise, is part of the program's control structure, the problem becomes even more complicated. The consequence of these observations is that success in a closed system comes from careful domain restrictions.

Since XREP is "domain-independent," it addresses a different class of problems than those normally associated with problem solving. Sussman tried to make this same distinction within HACKER by keeping general "programming knowledge" separate from specific "block world" details. The errors which XREP handles are associated with program writing flaws which are detectable at execution time, not those which might require problem solving in the translation phase in order to just get a program statement(vi).

The philosophy within XREP is to assume that the system can "help" detect errors and make strong, meaningful suggestions for correcting them. Of course, accepting advice from a human in an interactive environment about the presence or source of a bug should be welcome, even if the system might be better equipped to fix it. The situations which follow in succeeding sections take that approach: XREP finds a potential bug and offers a solution which the user can reject (causing backtrack) or accept.

5.7 *UNBOUND VARIABLES*

The first problem to be analyzed is the case of the unbound variable. The situation is quite common: a variable does not point to bound value. Consider the PLX program in Figure 37.

(vi) See [BALZER 74b] for a description of a Model Completion task within an Automatic Programming system which addresses that translation problem.

```
A :=  B , C
B :=  (GENMEM DIEVAL)
C :=  D , E , F
D :=  (TERMINAL 'IN D')
E :=  (TERMINAL 'IN E LOOKING FOR' DIEVAL.-
F :=  (TERMINAL 'IN F')
```

Figure 37. Unbound variable example

If this program is run, its PEG just before executing the TERMINAL statement of E will look like that in Figure 38.

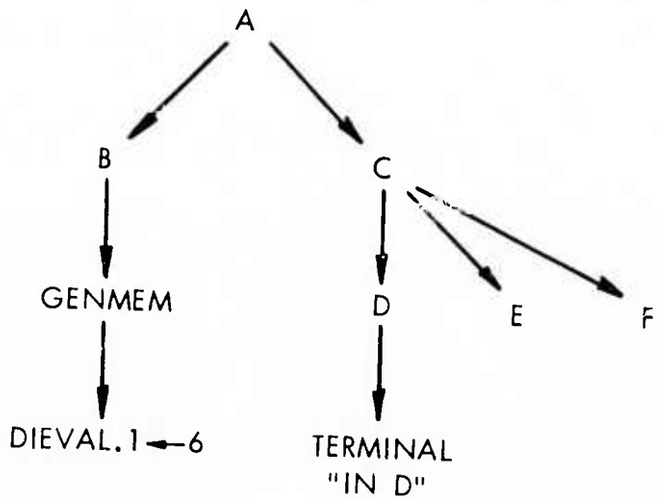


Figure 38. Partial PEG for program in Figure 37

The TERMINAL event in E accesses DIEVAL.-1. However, since no such binding exists in the context path of E, an unbound variable results. If the program has no sequencing problems, and if no spelling errors exist, there must be a fault in its structure. In this example the context path of E must be reconfigured to include an event with the appropriate typed variable generated in it. If such an event exists, there is always a way to accomplish this reconfiguration, though it may lead to other problems. Figure 39 shows a repaired version of the Access Graph of Figure 38.

The problem stems from the production rule

A := B , C

Presumably the error occurred because event C is erroneously protected from event B because of the event separator ",". To produce the proper Access Graph the rule should read:

A := B -> C

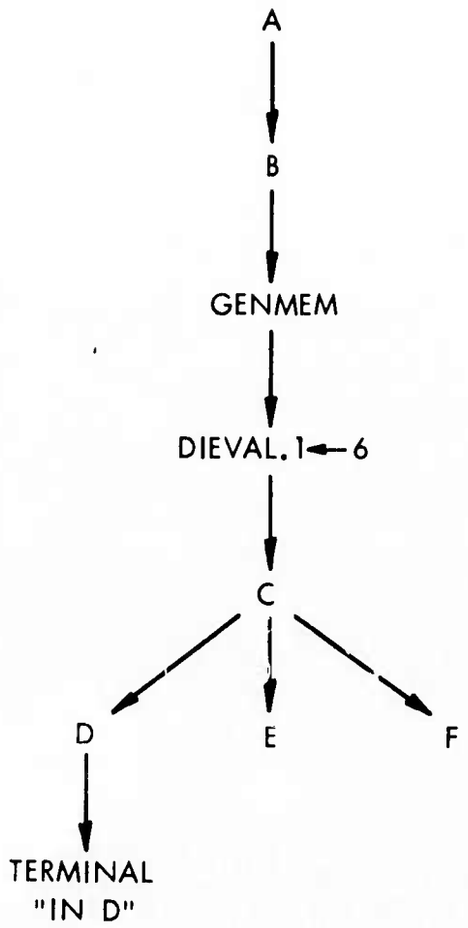


Figure 39. Repaired Access Graph

This example simplifies what has actually happened because the flaw shows up so vividly in the production system. In generating a program in a top-down manner a designer defines different levels of abstraction from which his program can be viewed. Each level is complete as a "machine" assuming the right primitives(vii). If a structural flaw like this one occurs, it is probably a case of a machine needing access to one which is at the same level. In this example, the hierarchy of computation makes the process B a black box to that of C. Yet it seems that it should be otherwise if the PLAYER.-1 request is valid. The reorganization described resolves exactly that.

(vii) This is, of course, Dijkstra's simile in his *Notes on Structured Programming* [DAHL 72].

The procedure to repair this problem will be called the NO-BIND algorithm. Its description is given in terms of the PEG, not the Access Graph. Figure 40 shows the original PEG for this example, annotated by the local symbols of the algorithm.

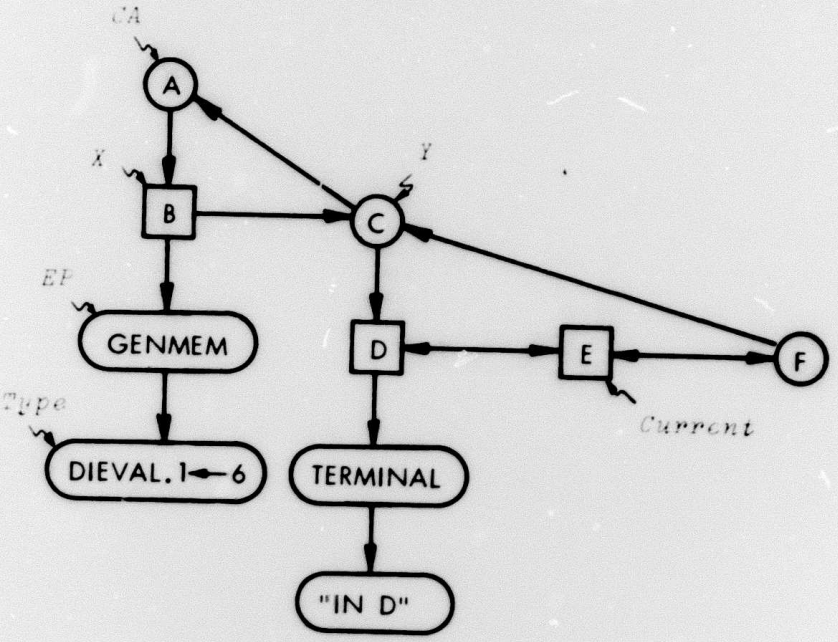


Figure 40. Annotated PEG for program in Figure 37

The algorithm, invoked when an access from event CURRENT to TYPE fails, is as follows:

NOBIND (TYPE, CURRENT)

1. Find a previous event, EP, which has an instance of TYPE in it.

2. Find the common rule ancestor, CA, to both CURRENT and EP.

3. Inspect the production rule for CA for the form

   CA := . . . X , . . . Y . . .

   where X is in the context path of EP and Y is in the context path of CURRENT.

4. Change that production rule for CA into

   CA := . . . X -> . . . Y . . .

5. Back up the process to the event following X and continue execution from there.

In the example EP, the previous target event is the GENMEM. The common rule ancestor, CA, is event A. X is event B ; Y is event C . The rule change in Step 4 makes the B event in Figure 40 oval instead of rectangular, thus making the GENMEM accessible to event C and all its descendants.

A few other things should be stated about the algorithm. In Step 1 the context path of EP is not subsumed by the context path of CURRENT; otherwise TYPE.-1 would be accessible to it.

In Step 2 the common rule ancestor may be different from the common ancestor if viewing an Access Graph. The PEG, however, makes this distinction obvious. Note also that there may be more than one rule whose change might solve the problem. The algorithm picks the "nearest" (in terms of time sequence) first.

In Step 3 a rule like

CA := X , Z . . . Y

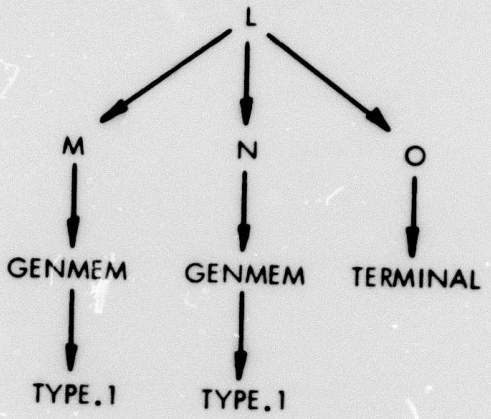
may be the culprit. However, changing it to

CA := X -> Z . . . Y

according to the algorithm so that Y gets access to X may be wrong because of Z's action. NOBIND fails here; a substantive structuring error has been made.

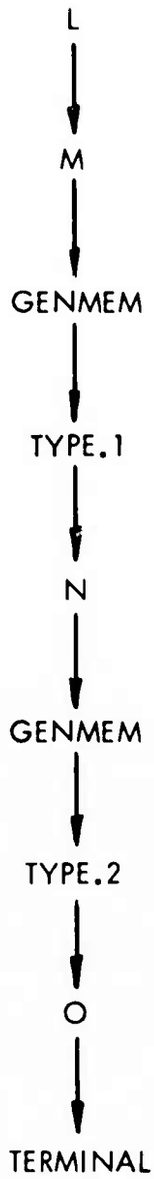
Step 5 actually is legitimate only if the instantiation of the CA rule in question is the first application of it. If the rule has been used before and is to be changed, the process must restart from its first application.

The NO-BIND algorithm, presented in terms of a TYPE.-1 failure, will also work if a TYPE.-2 request fails, requiring a reconfiguration which includes two instances of TYPE in the context path of the accessing event. Thus the situation in Figure 42 can be repaired by two applications of the algorithm to produce Figure 43.



L := M , N , O
M := (GENMEM TYPE)
N := (GENMEM TYPE)
O := (TERMINAL TYPE.-2)

Figure 42. A TYPE.-2 access failure



L := M -> N -> O
M := (GENMEM TYPE)
N := (GENMEM TYPE)
O := (TERMINAL TYPE.-2)

Figure 43. The repaired version of Figure 42

5.8 *WRONG BINDINGS*

The solution to the unbound variable error of the last section involved reorganizing access paths to make the proper information available. A byproduct of that reconfiguration is that all the information in the path of the once missing type becomes available to the original requestor. For example, in Figure 39, event E now has the DIEVAL binding in its path as required, but everything else generated by event B is available not only to E, but to D and F as well. This situation, which

may or may not be desirable, will be investigated further in Chapter 6, where an alternative method for solving those particular unbound variable errors will be suggested. With that in mind, the usefulness of the NOBIND algorithm might be open to question. However, the examples in this section present situations in which the algorithm performs substantive modifications.

The case of a wrong binding is more complicated than the unbound variable situation because of the larger variety of possible solutions and possible errors. In the unbound case not much can be done until some binding is found. The manifestation of the wrong binding error, however, is likely to occur during a match of a TERMINAL to the intention string, when much more information is available.

Consider the Access Graph fragment in Figure 44. If the rule for event F is

F := (TERMINAL 'MOVE' TYPE.-2)

and the current state of the intention string is

. . . (MOVE 4) . . .

then a failure occurs since the terminal output in F would be (MOVE 5). Assuming that TYPE.1 is supposed to be 5 and not 4 (perhaps verified by an earlier match in the intention string), then (unless a major flaw is responsible) the problem could be a simple error in the reference specification with the TERMINAL event, TYPE.-2 should be TYPE.-1.

The only debugging that can be done here is to check the other bindings in that access path and see if one works. If so, then some sort of confirmation is required and the change to the production rules is a simple one. If the access was done via some arbitrary expression like TYPE.(ADD X Y), then little help can be rendered.

Note that this simple technique can be tried if an access like PLAYER.-2 has no binding. If PLAYER.-1 does, it might be the correct one. Thus this method should be tried just before the NO-BIND algorithm is attempted.

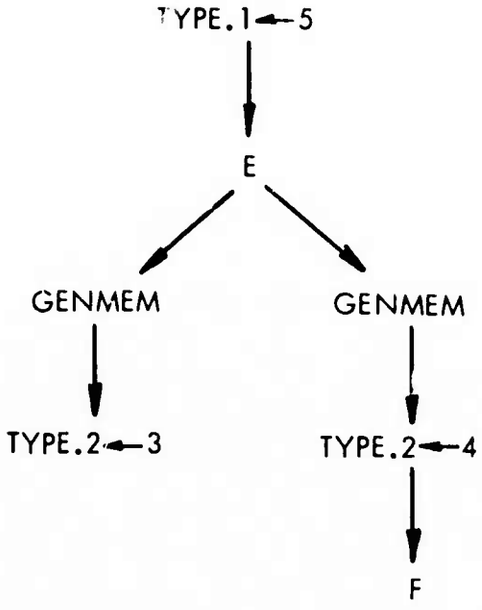


Figure 44. An access graph fragment

A more subtle case occurs when there is a structuring error. Suppose the rule for F in Figure 44 is

F := (TERMINAL 'MOVE' TYPE.-1 'AND' TYPE.-2)

and the current state of the intention string is

. . . (MOVE 4 AND 3) . . .

Then the failure, due to the mismatch of the TERMINAL output (MOVE 4 AND 5) and the intention string (MOVE 4 AND 3), is not so easily repaired. The search for the proper binding succeeds outside the current access path, indicating the need for an application of the NOBIND algorithm. The resulting Access Graph, shown in Figure 45, resolves the problem.

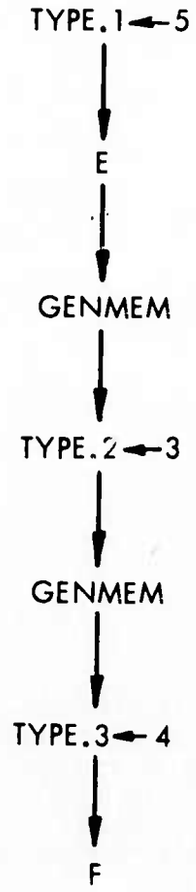


Figure 45. A linearized modification of Figure 44

The most substantive error situation is the inverse of the last example. With Figure 45 as the starting point and the same F rule as the preceding paragraph, the intention string

. . . (MOVE 4 AND 5) . . .

would provoke an error since (MOVE 4 AND 3) results from the TERMINAL event. The fix in this case -- to turn Figure 45 into Figure 44 -- comes from changing

E := (GENMEM TYPE) -> (GENMEM TYPE (-> F))

to

E := (GENMEM TYPE) , (GENMEM TYPE (-> F))

In other words, the event separator " -> " must be changed to a ",". This modification is just the opposite of what the NOBIND algorithm does, hence the inverse notion. A simple modification to that algorithm allows it to handle this error condition.

Notice that without the intention string to guide the results, very little could be done in these situations. The information conveyed by the matching process of TERMINALS to the intention string gives XREP's debugger a firm basis on which to diagnose the problem. Even if the situation is one which cannot be handled by XREP, the attempts it makes will at least be reasonable.

5.9 *RECURSION AND STRUCTURE FAULTS*

The last section described errors relating to binding issues which were quite clear-cut and needed debugging. Now a different kind of problem will be viewed, one in which there is "nothing" technically wrong. Instead the error will be strictly related to poor structure and an intuition about how a particular Access Graph should look.

This problem has a natural evolution and substantial basis arising out of a typical situation. An existing program needs to be modified. If the changes are made with only a local or narrow view of the problem, the resulting program can develop a "patched-together" look. In fact, HACKER gets exactly this kind of criticism from Sussman himself. His system creates a program in an evolutionary manner without the ability to step back and review it as a complete entity. This is obviously not a charge against HACKER, for that ability spans the entire intellectual programming discipline. Still, some "global" improvements can be made in a program if some assumptions are allowed.

The Backgammon example will again be the model. The actual rules for the start of a game need to be extended from those given on page 11. The complete statement is as follows:

> ***The game starts by having each player roll a die. The player with the largest value makes the first move, consisting of his roll and the roll of the other player. In case of a tie the value of the cube is doubled, and the process is repeated.***

The cube, initially 1, represents the value of the game in any arbitrary unit. Ignoring the tie condition for the moment, the program is shown in Figure 46.

```
BACKGAMMON := START , REST-OF-GAME
START    := (GENSEQ PLAYER T (-> ROLLDIE)) -> COMPARE
ROLLDIE := (GENMEM DIEVAL T) ->
                (TERMINAL PLAYER.-1 'ROLLED' DIEVAL.-1)
COMPARE := (INSERT PLAYER.(INDEX MAX DIEVAL)) -> FIRST-MOVE
FIRST-MOVE  := (TERMINAL PLAYER.-1 'MOVES'
                DIEVAL.1 FROM PLAYER.-1 'AND' DIEVAL.1 FROM
                PLAYER.(FIND (NEQ PLAYER PLAYER.-1)))
REST-OF-GAME := . . .
```

Figure 46. Backgammon program without tie condition

This program is essentially the same as the previous ones except for the added detail in the TERMINAL events. The ROLLDIE nonterminal now includes an observation of the rolling process, and the FIRST-MOVE TERMINAL makes the move explicit in terms of die values (as given in the English statement above).

Given the following initial AP1 top-level assertions

```
(ASSERT (AMO JOE PLAYER))
(ASSERT (AMO JOHN PLAYER))
(ASSERT (AMO 1 DIEVAL))
(ASSERT (AMO 2 DIEVAL))
(ASSERT (AMO 3 DIEVAL))
(ASSERT (AMO 4 DIEVAL))
(ASSERT (AMO 5 DIEVAL))
(ASSERT (AMO 6 DIEVAL))
(ASSERT (VALUE CUBE 1))
```

the execution of this program will match an intention string like

(JOE ROLLED 5) (JOHN ROLLED 3) (JOE MOVES 5 AND 3) . . .

My initial attempt to implement the tie condition quite naturally involved only the addition of another compare rule to handle the failure of the INDEX function's attempt to return a unique PLAYER when their die values are the same. The program, including this new rule, is shown in Figure 47.

```
BACKGAMMON := START , REST-OF-GAME
START   := (GENSEQ PLAYER T (-> ROLLDIE)) -> COMPARE
ROLLDIE := (GENMEM DIEVAL T) ->
                (TERMINAL PLAYER.-1 'ROLLED' DIEVAL.-1)
COMPARE := (INSERT PLAYER.(INDEX MAX DIEVAL)) -> FIRST-MOVE
COMPARE := (TERMINAL PLAYER.-1 'AND' PLAYER.-2 'TIE' ) ->
        (FUNCTION (ASSERT VALUE CUBE (ITIMES 2 (VALUE CUBE)))) ->
        (TERMINAL 'CUBE TURNED TO' (VALUE CUBE)) -> START
FIRST-MOVE  := (TERMINAL PLAYER.-1 'MOVES'
                DIEVAL.1 FROM PLAYER.-1 'AND' DIEVAL.1 FROM
                PLAYER.(FIND (NEQ PLAYER PLAYER.-1)))
REST-OF-GAME := . . .
```

Figure 47. Initial attempt to implement the tie condition

The new COMPARE rule has four parts. The first TERMINAL is an announcement that the two players tied. The FUNCTION statement asserts to the AP1 data base that the value of the cube is to be twice the old value, with the next TERMINAL expression announcing that fact. Finally the process is sent back to START to repeat. One simplification has been introduced. The expression (VALUE CUBE) is an abbreviation for

```
(FS* NUMBER (VALUE CUBE NUMBER))
```

This FS* function, first explained in Section 3.2, returns as its value the number which matches the data base assertion (VALUE CUBE NUMBER). Thus the FUNCTION statement asserts that the new value of the cube is twice the old value.

This new program was tested on two intention strings. With the original cube value at 1 (an initial assertion) the strings were

```
(JOE ROLLED 5) (JOHN ROLLED 3) (JOE MOVES 5 AND 3) . . .
```

and

```
(JOE ROLLED 4) (JOHN ROLLED 4) (JOHN AND JOE TIE)
(CUBE TURNED TO 2) (JOE ROLLED 5)
(JOHN ROLLED 6) (JOHN MOVES 6 AND 5) . . .
```

The first string is the original one; the second reflects the tie condition. In trying them out with the new rules it was rewarding to see everything work perfectly, at least until I looked at the Access Graph for the second intention string. Figure 48 shows in abbreviated form what actually happened.

I did not expect generation numbers for PLAYER to be 3 or 4. They became that high because the second GENSEQ had the first GENSEQ in its access path. Working backwards from the Access Graph, I realized that the proper structure should be like the one shown in Figure 49.

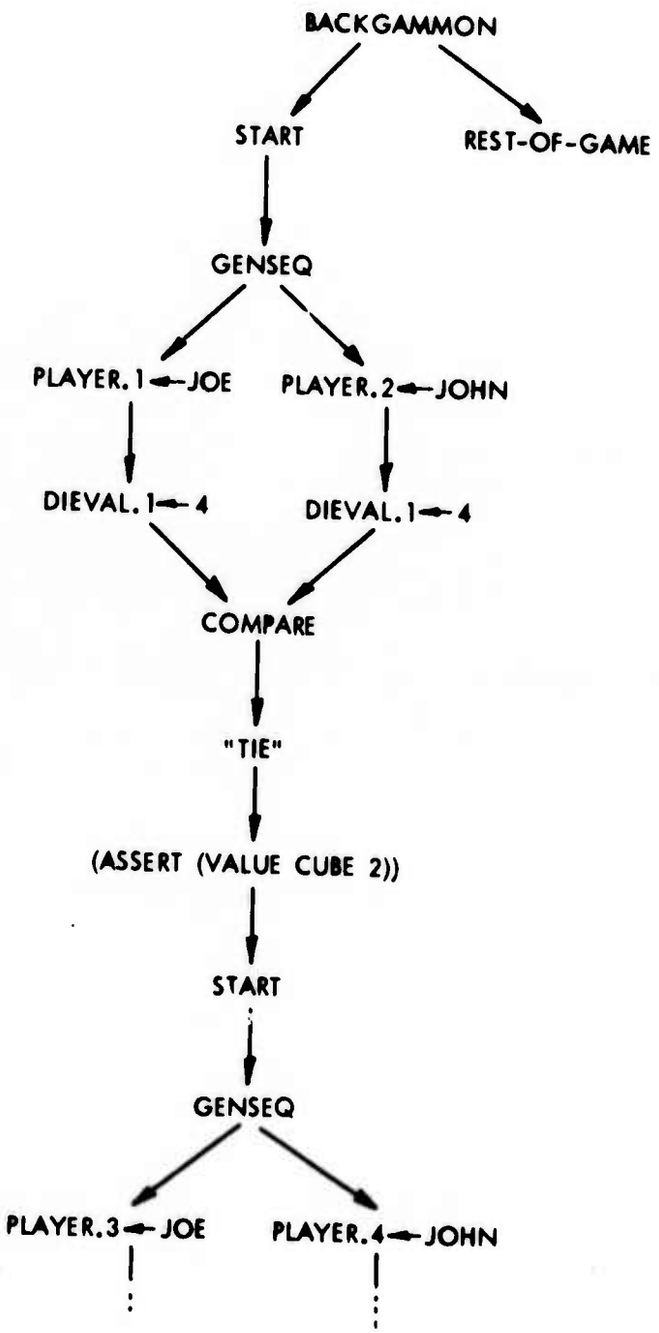


Figure 48. The Access Graph for the tie example

The recursion in Figure 48 occurred at too low a level, making irrelevant information available to the second die-rolling node. In Figure 49 that situation was remedied by making the recursion occur higher than the GENSEQ. This fix requires the presence of a new simple event, INITIAL-MOVE, on which to manage the recursion. A new set of rules which embody the fixed PEG is displayed in Figure 50.

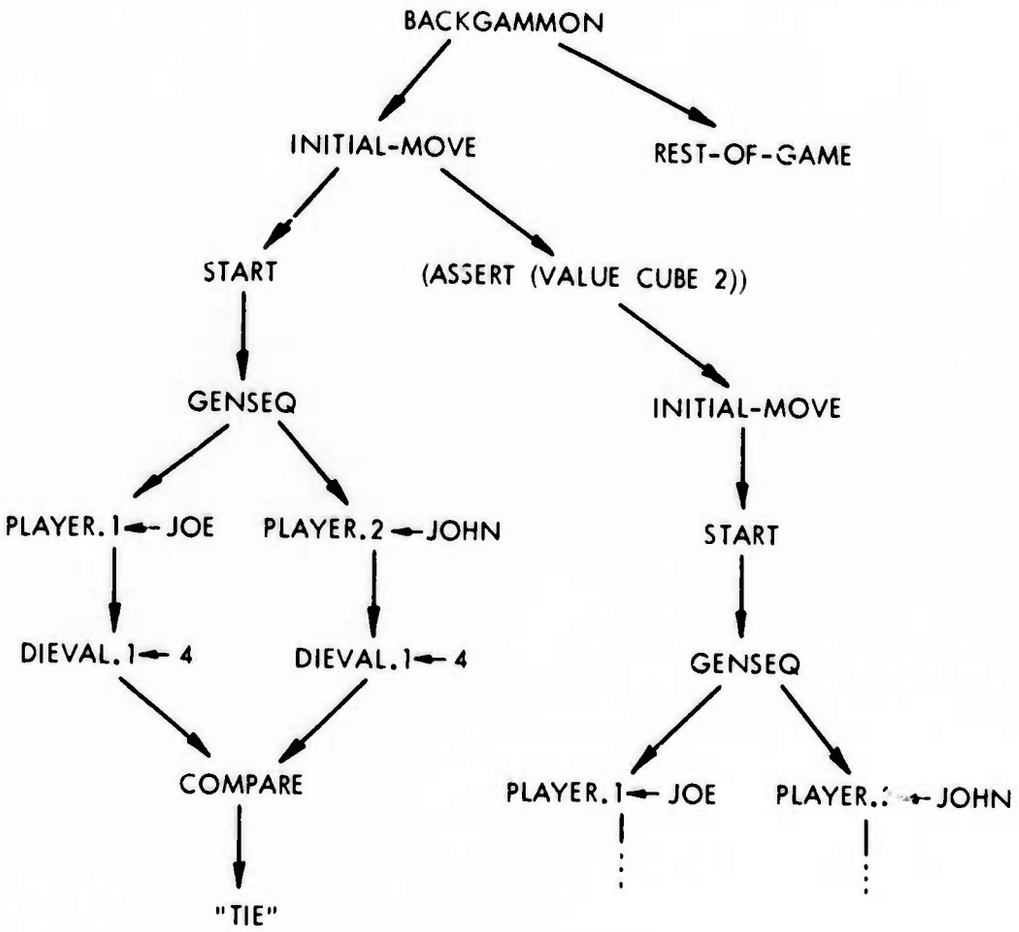


Figure 49. Corrected Access Graph for the tie example

```
BACKGAMMON := INITIAL-MOVE , REST-OF-GAME
INITIAL-MOVE := START
INITIAL-MOVE := START ,
     (FUNCTION (ASSERT (VALUE CUBE (ITIMES 2 (VALUE CUBE))))) ->
     (TERMINAL 'CUBE TURNED TO' (VALUE CUBE)) -> INITIAL-MOVE
START   := (GENSEQ PLAYER T (-> ROLLDIE)) -> COMPARE
ROLLDIE := (GENMEM DIEVAL T) ->
                (TERMINAL PLAYER.-1 'ROLLED' DIEVAL.-1)
COMPARE := (INSERT PLAYER.(INDEX MAX DIEVAL)) -> FIRST-MOVE
COMPARE := (TERMINAL PLAYER.-1 'AND' PLAYER.-2 'TIE' )
FIRST-MOVE  := (TERMINAL PLAYER.-1 'MOVES'
                DIEVAL.1 FROM PLAYER.-1 'AND' DIEVAL.1 FROM
                PLAYER.(FIND (NEQ PLAYER PLAYER.-1)))
REST-OF-GAME := . . .
```

Figure 50. Corrected Backgammon rules

The important differences in this new set of productions appear in the rules containing INITIAL-MOVE. The first INITIAL-MOVE rule anticipates the non tie condition, while the second is for the case when a tie occurs(viii).

The point of these rule changes is to force the "tie" recursion to occur at INITIAL-MOVE instead of START, so that each GENSEQ node will be unique within its access path, solving the problem of PLAYER generation numbers becoming higher than two.

Two issues now present themselves: How is this problem detected and how can it be fixed? If a program runs "correctly" (i.e., matches its intention string), then some explicit method must be devised to give the system the impetus to debug. One way is via constraints on the generation numbers. That is, in fact, how this problem originally presented itself.

The solution is to allow assertions which state the maximum limit for specific types. If that limit is exceeded, then the RESTRUCTURE algorithm (to be presented shortly) will be offered to the user. For the BACKGAMMON program, the assertion

(ASSERT (MAXSIZE PLAYER 2))

is added to the top-level set.

(viii) If the first INITIAL-MOVE rule is chosen and a tie game is being parsed, the failure of the rule does not occur in START (since a tie will still be declared by the second COMPARE rule) but in the failure of REST-OF-GAME to have the right beginning information: a unique player, two different die values, etc.

This kind of problem is difficult to formalize because of its wide range. When an implicit assumption -- whether built into a program or unstated by the user -- is violated, some system module must recognize the situation and act accordingly. Sussman's HACKER has a Critics Gallery which watches over the code generator so that when a proposed program statement is about to violate some condition, the appropriate critic will complain. This demon(ix) approach seems to imply that each assumption needs its own critic or expert, a possibility which may cause a computational explosion. Automatic Programming efforts will undoubtedly uncover many of these cases.

The RESTRUCTURE algorithm, invoked by the detection of a generation number overflow, follows. The PEG for this example, shown in Figure 51, is annotated by the locals of the algorithm, while some of its obvious links are not included. Remember that this PEG represents the Access Graph of Figure 48.

RESTRUCTURE (GUILTY)

1. Find the rule father, RF, of the GUILTY GENMEM or GENSEQ.

2. See if RF appears twice in the current context path. If not, the algorithm fails.

3. If so, find the grandfathers of each RF; call them GF1 and GF2. Let the two rules GF1 and GF2 have the following form:

   GF1 := X sep1 RF sep2 Y
   GF2 := Z sep MOVABLE-STUFF sep RF

4. Make the following changes to the program. Change the GF1 rule to be

   GF1 := X sep1 TEMP sep2 Y

   Change the GF2 rule to be

   GF2 := Z

   Insert the rules

   TEMP := RF
   TEMP := RF , MOVABLE-STUFF -> TEMP

(ix) A demon is a module which oversees execution and is invoked when some specific condition is met. PL/1 ON Conditions are an example of demon programs.

5. To compute which events comprise MOVABLE-STUFF, inspect the original GF2 rule from right to left, ignoring RF. For every Ei encountered do the following:

   a) If Ei makes a reference to a previous event which cannot be moved, then Ei cannot be moved. This decision may have to be delayed by step b.

   b) If Ei makes any assertions and has descendants which cannot be moved, it cannot be moved. Otherwise it can; this includes the case where dependents are waiting for a decision about Ei itself.

   c) The process stops when Ei cannot be moved.

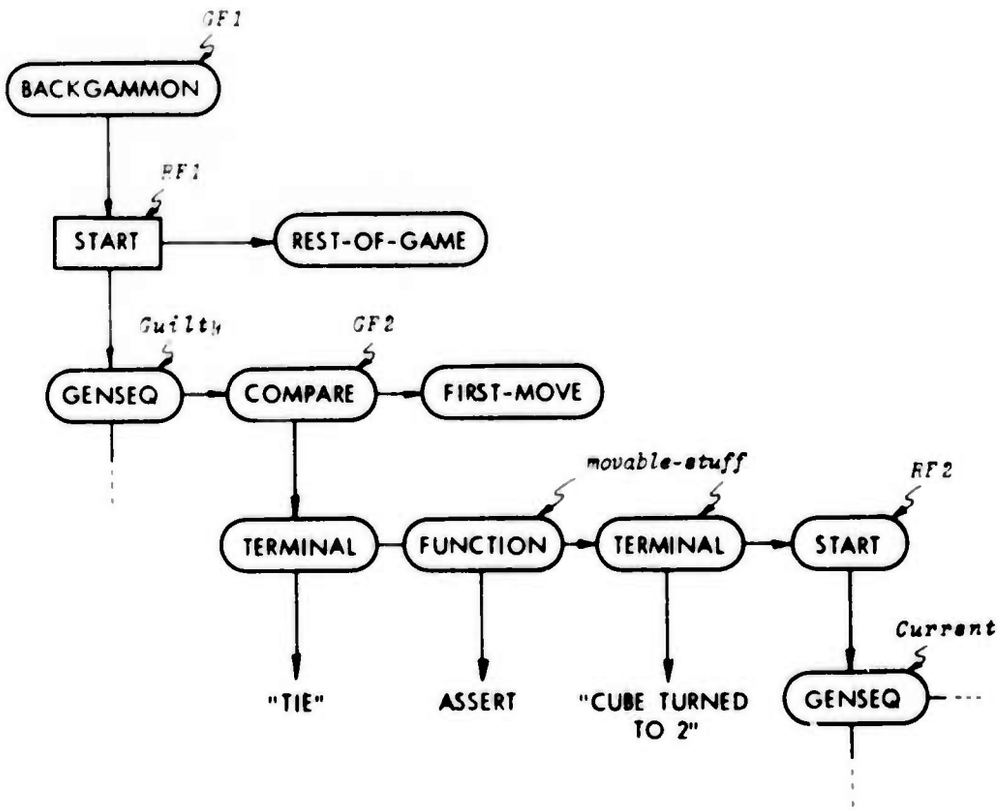


Figure 51. The annotated PEG for Figure 48

A few statements should be made about this algorithm before tracing it for the BACKGAMMON example. The MOVABLE-STUFF computation is done to find those events which will not be affected by the relocation performed by the RESTRUCTURE algorithm. The move is both aesthetic (since

MOVABLE-STUFF no longer need contend with the irrelevant information it was under) and functional (since its information might be needed for descendants of the new TEMP nonterminal). Notice that if a nonmovable event is needed under the second instantiation of TEMP, an irresolvable conflict will arise; XREP's NOBIND algorithm trying to undo the work of RESTRUCTURE, which will in turn be reinvoked by the same generation number overflow. This problem, unsolvable in the given framework, will be addressed further in the conclusion, Chapter 6.

The algorithm will now be traced.

In (1) START is the rule father (RF) of the GENSEQ which tries to generate PLAYER.3, violating the (MAXSIZE PLAYER 2) assumptions.

In (2) START does appear twice in the context path of the guilty GENSEQ.

In (3) GF1 is the BACKGAMMON nonterminal, while GF2 is the second COMPARE. The productions in question are

```
BACKGAMMON := START , REST-OF-GAME
COMPARE := (TERMINAL PLAYER.-1 'AND' PLAYER.-2 'TIE' ) ->
        (FUNCTION (ASSERT VALUE CUBE (ITIMES 2 (VALUE CUBE)))) ->
        (TERMINAL 'CUBE TURNED TO' (VALUE CUBE)) -> START
```

In (5), to determine MOVABLE-STUFF, inspect the three events of the COMPARE rule (ignoring START) from right to left. The (TERMINAL 'CUBE' . . .) event instantiates (VALUE CUBE), so a decision must be delayed until the source of (VALUE CUBE) is determined. In the FUNCTION statement an assertion is made which uses (VALUE CUBE). In this case (VALUE CUBE) is known to originate from the initial top-level assertions (i.e., nothing done by the program). Thus it can be moved, as can the delayed event (TERMINAL 'CUBE' . . .), which depends on it. The (TERMINAL PLAYER.-1 . . .) event instantiates two items -- PLAYER.-1 and PLAYER.-2 -- which are part of the original GENSEQ; it therefore cannot be moved, ending step 5.

Now step 4 can be completed. The four affected rules are

```
BACKGAMMON := TEMP , REST-OF-GAME
TEMP := START
TEMP := START ,
       (FUNCTION (ASSERT (VALUE CUBE (ITIMES 2 (VALUE-CUBE))))) ->
       (TERMINAL 'CUBE TURNED TO' (VALUE CUBE)) -> TEMP
COMPARE := (TERMINAL PLAYER.-1 'AND' PLAYER.-2 'TIE')
```

These rules are as advertised, TEMP taking the role of INITIAL-MOVE of Figure 50.

5.10 *PRECONDITIONS AND POSTCONDITIONS IN RECURSION*

The restructuring techniques derived in the last section can be applied to a class of algorithms exemplified by the instructions which might be found on a shampoo bottle.

1) Wet hair
2) Lather
3) Rinse
4) Repeat

Statement 4, the source of the problem, does not specify the starting iteration point; where should we repeat from? Notice also that no matter what iteration point is chosen, the lack of a terminating condition will cause this algorithm to loop forever. These kinds of flaws are typical of human-oriented instructions; the user is supposed to apply common sense to a situation to resolve ambiguities. In the shampoo example everyone would repeat from Step 2 since wetting already wet hair is nonsensical. Hardly anyone would lather up more than twice.

This problem, among others, is described by I. D. Hill in his paper about programming in English [HILL 72]. Its message is clear; Automatic Programming will have to cope with poorly specified algorithms and find methods to correctly translate them.

Given no other information, a likely guess for the starting point of the shampoo's "repeat" statement is the beginning one, "wet hair." Neglecting the infinite loop, that interpretation is incorrect because it includes a precondition, wetting the hair, in its main loop body. A program in PLX for this simple algorithm is shown in Figure 52.

```
SHAMPOO   :=  WET-HAIR -> LATHER -> RINSE -> REPEAT
WET-HAIR  :=  (TERMINAL 'WETTING HAIR')
LATHER    :=  (TERMINAL 'LATHERING HAIR')
RINSE     :=  (TERMINAL 'RINSING HAIR')
REPEAT    :=  SHAMPOO
```

Figure 52. The shampoo program

As written, the program will generate a terminal string like

(WETTING HAIR) (LATHERING HAIR) (RINSING HAIR)
(WETTING HAIR) (LATHERING HAIR) (RINSING HAIR)
(WETTING HAIR) . . .

However, a correct intention string for the shampoo exercise would be

(WETTING HAIR) (LATHERING HAIR) (RINSING HAIR)
(LATHERING HAIR) (RINSING HAIR) . . .

The detection of the precondition error in this simple example from the correct intention

string is not complicated. If a TERMINAL generates a nonmatching string which has occurred before, the precondition error might be hypothesized. If the program is allowed to continue, and the next TERMINAL does match, then the hypothesis is strengthened. In the shampoo program the second (WETTING HAIR) string does not match the intended (LATHERING HAIR). If continued, the program will however generate (LATHERING HAIR) as desired. The precondition problem seems to be at fault(x).

The REMOVE-PRECONDITION algorithm to be presented shortly assumes that the precondition event, PRE-EVENT (the second (WETTING HAIR) string), and the desired event, DESIRED-EVENT (the second (LATHERING HAIR) string), have been identified. The algorithm and an annotated abbreviated PEG follow.

REMOVE-PRECONDITION (PRE-EVENT DESIRED-EVENT)

1. Find the common rule father, RULE-FATHER, for PRE-EVENT and DESIRED-EVENT.

2. Check if RULE-FATHER's first son, PRE-FATHER, generated PRE-EVENT. If not, algorithm fails.

3. Check if RULE-FATHER has already appeared in the PEG. If not, fail.

4. Take the rule in question

   RULE-FATHER := PRE-FATHER sep rest-of-rule

   and rewrite it as the pair

   RULE-FATHER := PRE-FATHER sep TEMP
   TEMP := rest-of-rule

5. Replace instances of RULE-FATHER in the right hand side of all production by TEMP.

(x) Since having a postcondition in the main loop is exactly symmetric to the precondition case, it will not be discussed other than in this footnote.

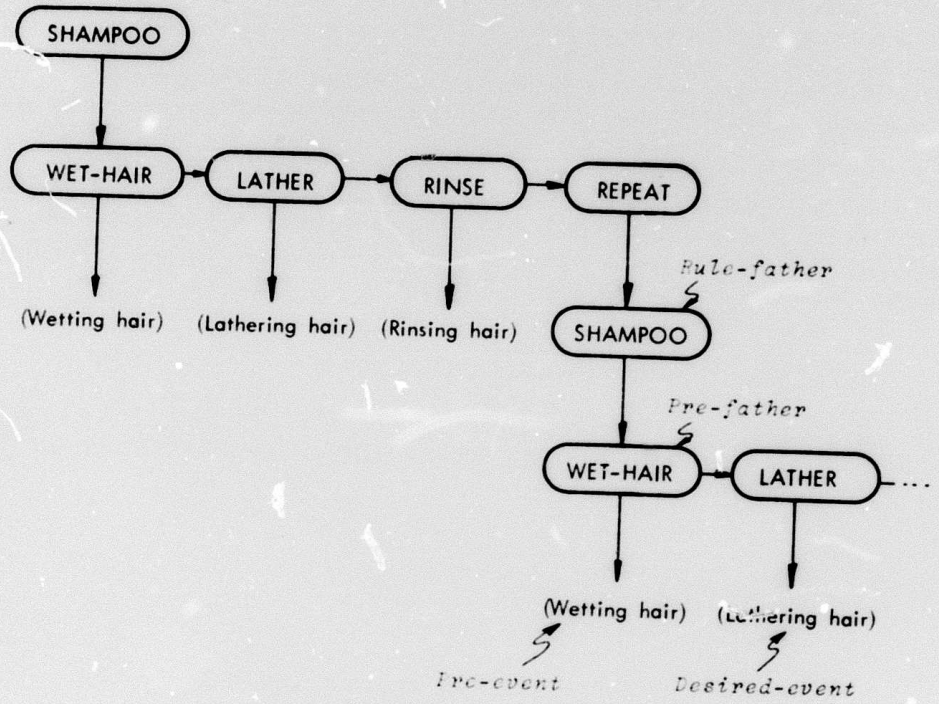


Figure 53. Annotated PEG for shampoo program

The effect of this algorithm on the SHAMPOO program is shown in Figure 54. The first SHAMPOO production sets up the execution with the precondition removed from the loop, now managed on TEMP. The REPEAT rule passes execution to the correct iteration point.

```
SHAMPOO   :=  WET-HAIR -> TEMP
TEMP      :=  LATHER -> RINSE -> REPEAT
WET-HAIR  :=  (TERMINAL 'WETTING HAIR')
LATHER    :=  (TERMINAL 'LATHERING HAIR')
RINSE     :=  (TERMINAL 'RINSING HAIR')
REPEAT    :=  TEMP
```

Figure 54. Fixed shampoo program

Unfortunately, the situation is much more difficult than has been described. The algorithm works only when PRE-EVENT and DESIRED-EVENT can accurately be determined. In the simple case of the SHAMPOO program that determination was easy, but consider the general situation. The event named PRE-FATHER (WET-HAIR in the SHAMPOO program) could expand into a series of TERMINALS, not just one as in the example. If so, DESIRED-EVENT will not appear for an indeterminate amount of time, making its identification tenuous. If the environment for the second, incorrect PRE-EVENT, causes a change in its evaluation, then it may not even be detectable that this situation has arisen.

Also the presence of the undesired PRE-EVENT may change the evaluation environment for DESIRED-EVENT, making it unmatchable with the intention string if execution is continued.

Once the two key events have been located, the rest follows. This situation is similar to some of the problems faced by Green and Waldinger in their automatic program generation work [GREEN 74]. Their research investigated generating programs from various input descriptions, I/O pairs, ellipses, etc. One of the examples is the following

INPUT		OUTPUT
(A B C D)	->	((A B) (A C) (A D) (B C) (B D) (C D))

The intended program is, of course, supposed to generate all 2-tuples from the input list. The first step of their system tries to identify the recursion point in the output list. Once (A B), (B C), and (C D) are located, the correct program will be generated. If this induction process fails, little else can be done.

The same is true for the REMOVE-PRECONDITION algorithm. Until this situation can be resolved by better debugging mechanisms, implementation of the algorithm is impractical. Some of the necessary improvements will be discussed in Chapter 6.

5.11 *RESOLVING PRONOMIAL REFERENCES*

The last error situation to be discussed, resolving pronomial references, is common in natural language translation programs. Charniak states the general reference problem as follows:

> If I tell a computer "Jack has a top and Mary also has a top," to show a minimum of understanding the machine must realize that I have mentioned two different tops. It will need some way to distinguish the two objects internally, and since in both cases I used the phrase "a top," the English description will not suffice. We will assume that the objects are represented by two distinct symbols, say, TOP1, and TOP2. Unfortunately, when people speak, they don't refer to TOP2, they say, "Mary's top," or "the top Mary found in the woods." It will be necessary for our machine to take such English descriptions and decide which (if any) internal symbol is being referred to. This, simply, is the reference problem.(xi)

(xi) E. Charniak, "Context and the Reference Problem." *Natural Language Processing*, Ed. by R. Rustin (New York: Algorithmics Press), 1973, pg. 311.

INTENTIONS AND DEBUGGING

This issue has been addressed several times in this report in describing both PLX and access paths. However, when the reference is some pronoun form, the reference problem becomes more difficult, since less information is present.

In most natural language programs, resolving a pronomial reference is an immediate concern; a statement is made, some response must be generated. Thus its techniques have a general problem-solving flavor. Particular solutions can be found in Winograd's famous blocks world program [WINOGRAD 71], McDermott's data base debugging program [McDERMOTT 74], Charniak's work referenced above, as well as others. Each must solve his problem given whatever information he currently has. In McDermott's case he will mark a tension point in his data base until a troublesome reference can be resolved, since the data base's consistency is his responsibility, not answering immediate questions.

The pressure within an interactive Automatic Programming system is different. First, an unresolved reference can be disambiguated by the user if the system asks. However, the fluency of man-machine dialogue will suffer if every pronomial reference is questioned. Secondly, the goal is not question answering, but generating programs. For XREP this means providing help in a representational way together with debugging possibilities.

Once again, the intention string is the key. The situation is as follows: (1) the Automatic Programming translator cannot resolve a particular reference; (2) instead of failing it delays a decision by substituting a specification like "?X" for the unknown reference; (3) XREP tries to resolve "?X" during execution by using information in the intention string.

Since a PLX typed variable has two parts, the reference probably can appear three ways: TYPE.?X, ?T.EXP, and ?T.?X. The first might correspond to an ambiguous reference like "the player," when more than one exists. The second could result from a statement like "the one with the largest value." The last is the worst case coming from an "it"-type reference.

The problem is solved as follows:

1. If the first position is unknown, i.e., it equals ?T, each possible type can be hypothesized to see if the reference makes sense. This step will be unnecessary if a TERMINAL which will resolve the problem is encountered first. If a COND or other event uses this variable first, a choice point based on possible types is set up.

2. If a TERMINAL which uses this variable is encountered and the intention string matches other than this unknown variable, the proper value for ?T can be deduced easily by finding the type of the corresponding object in the intention string.

3. If no resolving TERMINAL is found but the program concludes correctly, the

hypothesized type assigned in Step 1 is assumed correct and inserted in the program.

4. When the second position of the typed variable is ?X, XREP must also find the correct binding and insert the proper count in place of ?X.

An example can be given from the BACKGAMMON program. Suppose that the first variable, PLAYER.-1, in the FIRST-MOVE product was unknown, i.e., ?T.?X instead as in

FIRST-MOVE := (TERMINAL ?T.?X 'MOVES' . . .)

If the intention string for the program was

(JOE ROLLED 6) (JOHN ROLLED 1) (JOE MOVES 6 AND 1)

then the first two expressions will match routinely. But when FIRST-MOVE generates (?T.?X MOVES 6 AND 1), the reference apparatus is needed. ?T.?X is matched to JOE, whose type is found to be PLAYER. The search up the access path for a PLAYER bound to JOE shows it to be the first one found. Thus PLAYER.-1 resolves the reference and is substituted in the program.

Though this example and process are not particularly profound, several interesting points can be made.

- *This use of the intention string makes simple reference problems simple to solve.*
- *P1.X's typed variable, T.X, gives the reference problem a natural interpretation in either the T or X component.*
- *The access path search emphasizes how this dynamic, run-time approach simplifies the task.*

6. CONCLUSIONS AND FUTURE DIRECTIONS

This report has investigated some representational issues for both writing and analyzing programs within a hypothetical Automatic Programming framework. The motivation for many of the forms and analyses originated in an attempt to deal with situations likely to arise when a human describes an algorithm to a computer. Within this paradigm the goal that the programming language should mirror natural language methods whenever possible accounts for the production system approach to PLX, its data types, and structured organization. Similarly, the problems addressed by the various debugging discussions were meant to model common situations which, though natural in human communication, are imprecise or ambiguous in computer terms. The rest of this chapter, divided by these language and debugging goals, will review this report's accomplishements, and suggest future directions, while identifying problems discovered during the course of the research.

6.1 THE PRODUCTION LANGUAGE

My original thought about a target language for Automatic Programming leaned toward finding a process representation which was both machine oriented and had a "programming" flavor. A production-type language fulfilled both criteria, satisfying my initial goal. Next, I had to augment the standard notion of a production language with capabilities I envisioned necessary for the Automatic Programming task: intention strings, data generators, execution history, etc. With each added capability, PLX took on a more important role in the project than I envisioned, important enough to warrant an appraisal of PLX as a language development.

It has been said that a new programming language must contribute an order of magnitude more conceptual power to gain acceptance as a new vehicle [WILE 73]. That measure is hard to apply to PLX, since it was not designed to be used by human programmers; an appropriate alternative benchmark has not yet been established.

What can be measured is the effectiveness of PLX to deal with its three main issues:

1. PLX as a language.
2. The control structure of PLX.
3. The data generation and access mechanisms.

Language Aspects

First of all, programs can be written in PLX. This observation reacts to the range of processes for which PLX is intended, i.e., those which Automatic Programming might attempt to write. Though only a few programs (actually segments) were presented, others not included in the paper were written to insure that the language constructs used were adequate, and those needed were easily implementable as well as consistent with the formalism. The BACKGAMMON segment was sufficiently complex to test the adequacy of many of my representation goals: heterarchical organization, natural data referencing, and so forth. If an Automatic Programming system is built around a language like PLX, the present research will provide some, but not all, the inputs necessary to configure the best target language. Other language goals not tested by this study, like the coherency of large PLX programs, efficiency, and optimization, also need to be studied before any final conclusions can be made. Designing a computer language is an evolutionary process; PLX, as described here, is a first pass.

Control Structure in PLX

The control structure of PLX is difficult to evaluate. The clarity in passing control from event to event, and the simplicity of picking production rules and appending them to the PEG, added substantially to understanding the behavior of a PLX program. However, the backtrack mechanism for driving the productions was a mixed blessing. Though it allowed a "successful" execution to be found without worrying about false paths, it came at the expense of making firm commitments in the error detection phase impossible. While this is not the first time general backtrack has caused problems(i), several distinctive situations did arise.

If a top-down, syntax-driven parser picks a production which causes a failure, the process is backed up to the bad choice and a new rule is chosen. However, consider the PLX possibility with the two rules

```
A := (COND pred) -> B
A := C
```

An ALGOL interpretation of this production pair might be

```
IF pred THEN B ELSE C
```

If the predicate in the COND event fails, the action is clear, choose the A := C rule. But what if the predicate test passes, B is entered, and a failure occurs; should we backtrack, as PLX usually does now, or debug? In the ALGOL case, once B is entered, C is never again considered in this iteration; in PLX the failure in B is ambiguous. This is a general issue with generative systems: how is backtracking prevented when real errors are present? The utility of the TERMINAL statements can be seen in this situation: Event B can output enough information to identify the manifestation of a possible error. This solution will not take care of all cases, but it is certainly adequate for a

---

(i) See ***From PLANNER to CONNIVER -- A Genetic Approach*** [SUSSMAN 72] for a similar situation.

substantial set. In no case did I ever confuse a real error with a normal backtracking situation (while watching the process at a terminal); whether a novice will be so adept is a different question.

A different, less substantial issue in this area concerns varied control structures for producing iteration. Between the GENSEQ and production rule recursive control mechanism, all iterations are possible. However, large ones could easily cause an explosion in PEG size. One solution might incorporate some loop skeleton or frame structure to represent a loop, with local changing values updated as indicated. More important is that such a frame may be the basis for understanding loop execution and their associated problems. In any case, some more specific loop structures are needed in a complete Automatic Programming target language.

Data Generation and Access Mechanisms

The data methods in PLX attempt to incorporate a problem-solving function as a syntactic language device. The access path searches, the relative reference types, and the generators all try to retain and use dynamic information in a manner natural to English. Keeping these functions in the language allows a flexibility of expression and an information gain which would be lost if all references were resolved (or attempted to be resolved) at translation time. The major contribution of PLANNER [HEWITT 72] was the formalization of powerful problem-solving tools, like backtrack and pattern invoked procedures, into a coherent language. PLX does the same with its typed variables.

The basic idea in accessing dynamically produced data is that data has a position within an execution that carries information which can be exploited to the system's benefit. For example, the reference

DIEVAL.1 FROM PLAYER.-2 FROM ROLLDIE

from page 55 not only shows how an ambiguity can arise, but gives a graphic interpretation of it as well. If the manifestation of all problems were so explicitly capturable, debugging efforts would be minimized.

A different situation, in which the spatial nature of data is natural and easy to accept, exists when complex linkage between two data items is necessary before an association between them can be made. For example, in a card game, a request for all the Kings a particular player holds might be something like (HAS JOHN KING). Requests like this generally fail because players have cards and cards not players have rank. So, some inference mechanism must find the appropriate linkage to make the request legitimate. In PLX if the cards are generated after a player, the player and the cards are associated merely by being adjacent in the same access path. This association, powerful, simple, and intuitive, is easy to express in PLX.

The spatial positioning of data can also be applied in a manner not currently allowed in XREP. Recall that when a typed variable is unbound, the NOBIND algorithm tries to find the required instance, then reconfigures the access paths so the request succeeds. In solving the problem, the algorithm makes available not only the required data, but everything else which may be in its access

path as well. This linearization process may or may not cause future errors, but generally it seems undesirable if an alternative exists.

One suggestion called for an INSERT-like primitive which could move data from one access path to another when the situation arises, much like giving an access path a value (or values) and backing them up high enough in the PEG so others have access to them. This idea is intriguing because of its simplicity. Since it can be done with some syntactic primitive much like INSERT, the debugging situation can easily be envisioned. My concern is with how natural such a construct is; I do not think English has a corresponding construct.

A different solution is to permit declarations in the language. If a reference across access paths is detected, the declaration of the required variable is moved high enough to make it properly accessible. Even though declarations are not part of natural language, including them a language like PLX may be perfectly sound; certainly they simplify problems like these, though perhaps only locally.

In most common reference situations PLX was shown to have a form naturally corresponding to English. The FROM reference, and the INDEX and FIND functions handled a variety of common situations. Still some improvements are possible. One is to allow a GENSEQ or GENMEM to produce previously generated items. Recall that its members now come from assertions in the global associative API data base (See page 31) This facility would give an interpretation to a statement like "generate a sequence of all the existing players who have a King in their hands."

Also needed is a precise formalization of the capabilities of these methods. The INDEX and FIND functions fall into this category. What are they capable of retrieving? What situation will cause trouble? Are other functions required? Can a precise explanation be formulated for the anomalous situations raised by the generation number problem in Figure 27 or the accessing problem in Figure 29? These questions are hard to address in the present configuration of PLX because the exact range and domain of the functions are unknown. The lack of formality is not disturbing yet; experience with a particular method is generally required before formalization is possible. Some experience has been gained in the experiments of this dissertation; more is needed for proper understanding.

6.2 *THE PEG, INTENTIONS, AND DEBUGGING*

The three items of this section are grouped together because of their interdependence: The PEG, as the major information structure relating execution to the production rules, guides the debugging process, which is in turn often activated by intention string failures. Each will be reviewed as an entity and as a part of XREP.

CONCLUSIONS AND FUTURE DIRECTIONS

The PEG

The need for all execution information within a debugging system is mandatory. The PEG not only has that information, but has it structured to correspond to the original program. The explicitness of this correspondence was a design goal from the start of the project -- one that proved its worth throughout all the debugging exercises. It was rewarding to find that the PEG could maintain that relationship while yielding to a fairly straightforward implementation.

The Intention String

The intention string mechanism, on the other hand, is less well understood. It represents an informal attempt to provide XREP with parsing capabilities; the intention strings comprise the "language" of acceptable output for a program in much the same manner that formal languages can be described by a BNF "program." In the latter case the output strings are programs themselves; in the former the strings are merely some observation of a program's behavior.

The parsing ability is a natural way of matching three redundant specifications: the program, the TERMINAL events, and the intention string. It is interesting that the redundancy of English dialogue(ii) constantly gives the listener reassuring feedback to aid understanding. Old programming languages never considered incorporating such information(iii).

One of the complaints about the first-order predicate calculus as an assertion language is the stifling amount of detail with which the user must contend. A similar argument might apply to the intention string mechanism. Whether or not it is too bulky for practical use remains to be seen, but its flexibility will allow any level of detail by way of the TERMINAL events. In fact, different parts of a program can mix the desired detail in order to focus on specific modules or to deemphasize modules which have been shown to be reliable. Without that capability any specification mechanism is suspect. Research in this area is just beginning and is sure to have an impact on future programming practices.

Debugging

This dissertation has placed XREP into a general program-writing environment (whether an Automatic Programming type or an INTERLISP type is not important for the moment). Thus its

---

(ii) See [HALPERN 66] for a discussion of the role of redundancy in English and its possibilities in computer languages founded on English principles.

(iii) A new ALGOL dialect, ALGOL W [SITES 72], has incorporated an ASSERT statement, while the program proving system described in [GOOD 75b] has augmented PASCAL [JENSEN 73] with assertion statements, but only for their own use.

techniques are more general and less deep than those found in more domain-specific programs. The problems addressed in Chapter 5 were not startling, merely common, and if they could be solved without a great deal of commotion, the programming process becomes that much easier. Sackman describes this phenomenon in terms of a push-pull model, a programmer and computing system interact in trying to generate programs: the system pulls the programmer to a solution, while the programmer pushes in the same direction [SACKMAN 74]. The tendency in modern programming systems is to give the system more freedom in its work, i.e., it pulls the programmer nearer to the final solution before giving control back to the programmer. XREP's debugging algorithms are intended to work in this direction.

It is hard to evaluate my debugging algorithms because they depend so specifically on the representations offered by XREP, yet the problems they work on are general programming errors. Therefore, rather than dwell upon their effectiveness within XREP, I will discuss what makes them work and how they can be improved.

Without some form of expectation together with a complete execution history, an understanding system can only work in a superficial manner. However, much more is necessary before substantial analysis is possible. The most noticeable deficiency within XREP was in the lack of power of matching TERMINAL output strings to the intention string. Basically, the match is a one-for-one process which is not deep enough to take advantage of all the information present. The situations leading to the REMOVE-PRECONDITION algorithm of Section 5.10 point to some needs.

A sophisticated matcher could make all kinds of assumptions which individual debugging routines could access for relevance. All this work to identify the source of a bug is not simple, but necessary. When a match fails, the following should be attempted or questioned:

- How similar is the output to the target?
- Will some simple reordering within the string work?
- Has the output appeared before?
- Does the current output occur later in the expected output?
- Has some pattern developed which allows a system prediction?

The list is obviously not complete, but it is clear that in watching a program execute, a human asks exactly those kinds of questions in tracing down errors. A proper repertoire of similar techniques would even address difficult program sequence problems, where reordering program segments is the only solution.

In summary, if a bug is found and identified, automatic error correction is feasible given an environment as rich as the one present in XREP. Finding that bug is an order of magnitude (at least) more difficult. Continued research into classifying and theorizing about bugs can actually follow the design of a "library" of debugging techniques which might be activated by an on-line user as a tool once he has himself identified a bug.

REFERENCES

[BALZER 72]
Balzer, R.M. ***Automatic Programming.*** USC/Information Sciences Institute, RR-73-1, September 1972, **(draft).**

[BALZER 73]
Balzer, R.M. "CASAP: A Testbed For Program Flexibility." ***Proceedings of the Third International Joint Conference on Artificial Intelligence,*** Stanford University, August, 1973, pp. 601-605.

[BALZER 74a]
Balzer, R.M.; Greenfeld, N.R.; and Wilczynski, D. ***AP/1 - A Language for Automatic Programming.*** USC/Information Sciences Institute, RR-73-13, (in progress)

[BALZER 74b]
Balzer, R.M.; Greenfeld, N.R.; Kay, M.J.; Mann, W.C.; Ryder, W.R.; Wilczynski, D.; and Zobrist, A.L. ***Domain-Independent Automatic Programming.*** USC/Information Sciences Institute, RR-73-14, March 1974.

[BOBROW 74]
Bobrow, D.G., and Raphael, B. "New Programming Languages for AI Research." ***Computing Surveys,*** Vol. 6 (September 1974), pp. 155-174.

[BUCHANAN 69]
Buchanan, B.G.; Sutherland, G.L.; and Feigenbaum, E.A. "Rediscovering some Problems of Artificial Intelligence in the Context of Organic Chemistry." ***Machine Intelligence 5.*** Edited by B. Meltzer and D. Michie. Edinburgh: Edinburgh University Press, 1969.

[BUCHANAN 71]
Buchanan, B.G.; Feigenbaum, E.A.; and Lederberg J. "A Heuristic Programming Study of Theory Formation in Science." ***Proceedings of the Second International Joint Conference on Artificial Intelligence,*** Imperial College, London (September 1971), pp. 40-48.

[BUCHANAN 72]
Buchanan, B.G.; Feigenbaum, E.A.; and Sridharan, N.S. "Heuristic Theory Formation: Data Interpretation and Rule Formation." ***Machine Intelligence 7.*** Edited by B. Meltzer and D. Michie. New York: John Wiley & Sons, 1972.

[BUCHANAN 74]
Buchanan, J.R., and Luckham, D.C. ***On Automating the Construction of Programs.*** Stanford University, STAN-CS-74-433, 1974.

[CHARNIAK 73]
Charniak, E. "Context and the Reference Problem" ***Natural Language Processing.*** Edited by R. Rustin. New York: Algorithmics Press, 1973.

[DAHL 66]

Dahl, O.-J., and Nygaard, K. "SIMULA -- an ALGOL-Based Simulation Language." *Communications of the Association for Computing Machinery*, Vol. 9 (September 1966), pp. 671-678.

[DAHL 72]

Dahl, O.-J.; Dijkstra, E.W.; and Hoare, C.A.R. *Structured Programming*. New York: Academic Press, 1972.

[DIJKSTRA 72]

Dijkstra, E.W. "Notes on Structured Programming." *Structured Programming*. Edited by C.A.R. Hoare. New York: Academic Press, 1972.

[ELSPAS 1972]

Elspas, B.; Levitt, K.N.; Waldinger, R.J.; and Waksman, A. "An Assessment of Techniques for Proving Programs Correct." *Computing Surveys*, Vol. 4 (June 1972), pp. 97-147.

[HALPERN 66]

Halpern, M. "Foundations of the Case for Natural-Language Programming." *Proceedings of the Fall Joint Computer Conference*, Vol. 29 (November 1966), pp. 639-649.

[HEWITT 72]

Hewitt, C. *Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot*. Massachusetts Institute of Technology, AI-TR-258, 1972.

[GINSBURG 66]

Ginsburg, S. *The Mathematical Theory of Context-Free Languages*. New York: McGraw-Hill Book Company Inc., 1966.

[GOLDSTEIN 74]

Goldstein, I.P. *Understanding Simple Picture Programs*. Massachusetts Institute of Technology, AI-TR-294, 1974.

[GOOD 75a]

Good, D.I. "Provable Programming." *Proceedings of the International Conference on Reliable Software*, Los Angeles, April 1975, pp. 411-419.

[GOOD 75b]

Good, D.I.; London, R.L.; and Bledsoe, W.W. "An Interactive Program Verification System." *Proceedings of the International Conference on Reliable Software*, Los Angeles, April 1975, pp. 482-492.

[GREEN 74]

Green, C.C.; Waldinger, R.J.; Barstow, D.R.; Elschlager, R.; Lenat, D.B.; McCune, B.P.; Shaw, D.E.; and Steinberg, L.I. *Progress Report on Program-Understanding Systems*. Stanford University, STAN-CS-74-444, 1974.

[HILL 72]

Hill, I.D. "Wouldn't It Be Nice If We Could Write Programs in Ordinary English -- or Would It?" *The Honeywell Computer Journal*, Vol. 6, No. 2 (1972), pp. 76-83.

[JENSEN 74]
Jensen, K., and Wirth, N. *PASCAL -- User Manual and Report*, Vol. 18 of *Lecture Notes in Computer Science*, Edited by G. Goos and J. Hartmanis, Berlin: Springer-Verlag, 1974.

[JOHNSTON 71]
Johnston, J.B. "The Contour Model of Block Structured Processes." *Proceedings of a Symposium on Data Structures in Programming Languages*, University of Florida, 1971, pp. 55-82.

[LINGARD 72]
Lingard, R.W., and Wilczynski, D. "A Syntax Directed Approach for Handling Natural Language Relations." *Proceedings of the Association for Computing Machinery*, Boston, 1972, pp. 118-128.

[LINGARD 75]
Lingard, R.W. "A Representation for Semantic Information Within an Inference Making Computer Program." Unpublished Ph.D. dissertation, University of Southern California, 1975.

[MANN 73]
Mann, G.A. "A Survey of Debug Systems." *The Honeywell Computer Journal*, Vol. 7, No. 3, (1973), pp. 182-198.

[MARK 74]
Mark, W.S. *A Model-Debugging System*. Massachusetts Institute of Technology, MAC TR-125, 1974.

[McDERMOTT 74]
McDermott, D.V. *Assimilation of New Information by a Natural Language-Understanding System*. Massachusetts Institute of Technology, AI TR-291, 1974.

[MINSKY 72]
Minsky, M., and Papert, S. *Progress Report*. Massachusetts Institute of Technology, AI Memo 252, 1972.

[NEWELL 72]
Newell, A., and Simon, H.A. *Human Problem Solving*. New Jersey: Prentice-Hall, 1972.

[SACKMAN 74]
Sackman, H., and Blackwell, F.W. *Studies in Real-World Problem Solving With and Without Computers*. The Rand Corporation, R-1492-NSF, May 1974.

[SITES 72]
Sites, R.L. *Algol W Reference Manual*. Stanford University, STAN-CS-230, 1972.

[SUSSMAN 72]
Sussman, G.J., and McDermott, D.V. "From PLANNER to CONNIVER -- A Genetic Approach." *Proceedings of the Fall Joint Computer Conference*, Vol. 41 (1972), pp. 1171-1179.

[SUSSMAN 73]
Sussman, G.J. *A Computational Model of Skill Acquisition.* Massachusetts Institute of Technology, AI TR-297, 1973.

[TEITELMAN 72]
Teitelman, W. "Automated Programmering -- The Programmer's Assistant." *Proceedings of the Fall Joint Computer Conference*, Vol. 41 (1972), pp. 917-921.

[TEITELMAN 74]
Teitelman, W. *INTERLISP Reference Manual*, Xerox Palo Alto Research Center, 1974.

[WATERMAN 70]
Waterman, D.A. "Generalization Learning Techniques for Automating the Learning of Heuristics." *Artificial Intelligence*, Vol. 1 (1970), pp. 121-170.

[WATERMAN 73]
Waterman, D.A., and Newell, A. "PAS-II: An Interactive Task-Free Version of an Automatic Protocol Analyzing System." *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford University (August 1973), pp. 431-445.

[WATERMAN 74]
Waterman, D.A. *Adaptive Production Systems.* Carnegie-Mellon University, Complex Information Processing Working Paper 285, 1974.

[WEGNER 72]
Wegner, P. "The Vienna Definition Language." *Computing Surveys*, Vol. 4 (March 1972), pp. 5-63.

[WILE 73]
Wile, D.S. "A Generative, Nested-Sequential Basis for General Purpose Programming Languages." Unpublished Ph.D. dissertation, Carnegie-Mellon University, 1973.

[WINOGRAD 71]
Winograd, T. *Procedures as a Representation for Data in a Computer Program for Understanding Natural Language.* Massachusetts Institute of Technology, MAC TR-84, 1971

[WINSTON 72]
Winston, P.H. "The MIT Robot." *Machine Intelligence 7.* Edited by B. Meltzer and D. Michie. New York: John Wiley & Sons, 1972.

[WULF 73]
Wulf, W., and Shaw, M. "Global Variable Considered Harmful." *SIGPLAN Notices*, Vol. 8 (February 1973), pp. 28-34.

[YONKE 75]
Yonke, M.D. "A Knowledgeable, Language-Independent System for Program Construction and Modification." Unpublished Ph.D. dissertation, University of Utah, 1975.